Diseño e implementación en FPGA de un filtro de partículas para aplicaciones biomédicas

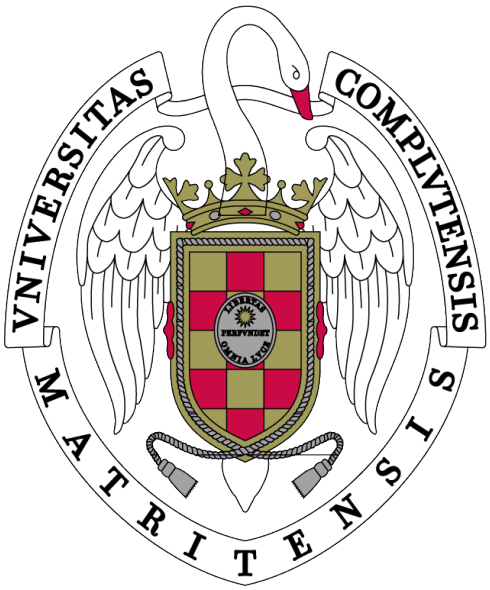
José Carlos González Salas

Máster en Ingeniería Informática

Facultad de informática

Departamento DACYA

Universidad Complutense de Madrid



Trabajo fin de Máster en Máster en Ingeniería Informática

Madrid, 04 de Septiembre de 2016

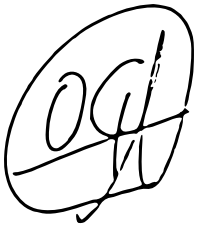
Convocatoria: Septiembre de 2016

Calificación: 8,7

Directores: Óscar Garnica Alcázar

Juan Lanchares Dávila

Curso Académico 2015-2016



Autorización de difusión y utilización

El abajo firmante, matriculado en el Máster de Ingeniería de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presenta Trabajo Fin de Master: Diseño e implementación en FPGA de un filtro de partículas para aplicaciones biomédicas, realizado durante el curso académico 2015-2016 bajo la dirección de Óscar Garnica Alcázar y Juan Lanchares Dávila en el Departamento de Arquitectura de Computadores y Automática, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en internet y garantizar su preservación y acceso a largo plazo.

Autor

José Carlos González Salas

Fecha

04/09/2016

Índice

Índice de figuras

Índice de tablas

Índice de abreviaturas

En este apartado se listan las abreviaturas y acrónimos que se utilizarán a lo largo del documento.

DM Diabetes mellitus.

OMS Organización Mundial de la Salud.

MCG Monitores Continuos de Glucosa.

PDF Función de Densidad de Probabilidad.

FK Filtro *Kalman*.

EFK Filtro *Kalman* Extendido.

FP Filtro de partículas.

AGE *Average Glucose Error*.

FPGA *Field Programmable Gate Array.*

FSM *Finite State Machine.*

Resumen

La diabetes mellitus tipo 1 (DM1) es una enfermedad crónica caracterizada por la incapacidad del páncreas de producir insulina. Esta hormona regula la absorción de la glucosa del torrente sanguíneo por parte de las células. Debido a la ausencia de insulina en el cuerpo, la glucosa se acumula en el torrente sanguíneo provocando problemas a corto y largo plazo, como por ejemplo deterioro celular.

Los pacientes con esta enfermedad necesitan controlar su glucemia (concentración de glucosa en sangre) midiendo la misma de forma regular e inyectándose insulina subcutánea de por vida. Para conocer la glucemia se pueden utilizar Monitores Continuos de Glucosa (MCG), que proporcionan el valor de la glucosa intersticial en un rango entre uno y cinco minutos. Los MCG actuales presentan los siguientes problemas:

- Por un lado, el sensor que lleva incorporado introduce ruidos asociados a la medición obtenida.
- Y, por otro lado, el sensor se degrada a lo largo de su vida útil, lo que dificulta la interpretación de los datos obtenidos.

La solución propuesta en este trabajo consiste en la utilización de filtros de partículas. Este tipo de filtros consta de cuatro fases: inicialización, predicción, corrección y *remuestreo*. Son capaces de identificar los estados ocultos del sistema (glucosa en sangre y degeneración del sensor), a partir de medidas indirectas del mismo (como por ejemplo la glucosa intersticial) teniendo en cuenta el ruido de las mediciones del MCG.

En este proyecto se va a aplicar un filtro de partículas de cuatro estados (glucosa, velocidad de variación de la glucosa, degeneración del sensor y velocidad de variación de la degeneración del sensor.). En primera instancia, se utilizará la herramienta *Matlab* para analizar el correcto funcionamiento de este algoritmo frente a los problemas mencionados anteriormente de los MCG. Y, en segundo lugar, se realizará una implementación *hardware* sobre una *FPGA*.

Palabras clave: *FPGA*, Filtro de partículas, Páncreas Artificial, Diabetes Mellitus, Diseño Hardware, Procesado de Señal.

Abstract

Mellitus Diabetes type 1 (MD1) is a chronic disease characterized by the pancreas inability to produce insulin. This hormone regulates the absorption of glucose from the bloodstream by cells. Due to the absence of insulin in the body, glucose builds up in the bloodstream causing problems in short and long term, such as cellular deterioration.

Patients with this disease need to control their glycemic (blood glucose concentration) by measuring it regularly and injecting subcutaneous insulin for life. In order to know glycemic can be used Continuous Glucose Monitor (CGM) which provide the value of interstitial glucose in a range between one and five minutes. Current CGM have the following problems:

- On one hand, the incorporated sensor introduces noise associated with the measurement obtained.
- And, on the other hand, the sensor degrades over its useful life, which makes data interpretation more difficult.

The solution proposed in this paper is the use of particle filters. This type of filter consists in four phases: initialization, prediction, correction and resampling. They are able to identify the hidden states of the system (blood glucose and sensor degeneration) from indirect measurements thereof (interstitial glucose) considering CGM noise measurements.

This project will apply a four states particle filter (glucose, glucose rate of change, sensor degeneration, and sensor degeneration rate of change). In the first instance, the Matlab tool will be used to analyze the correct functioning of this algorithm against the aforementioned problems of the CGM. And, second, a hardware implementation on an FPGA will be made.

Keywords: FPGA, Particle filter, artificial pancreas, Mellitus Diabetes, Hardware Design, Signal Processing.

Capítulo 1

1 Introducción

1.1 Motivación

La diabetes mellitus (DM) es una enfermedad metabólica producida por una secreción deficiente de insulina, lo que produce un exceso de glucosa en la sangre [1]. La Organización Mundial de la Salud (OMS) reconoce tres tipos de diabetes. En primer lugar, la DM tipo 1 en la que no se observa producción de insulina. En segundo lugar, la DM tipo 2 en la que el cuerpo sí produce insulina, pero, o bien no en cantidad suficiente, o bien no es capaz de aprovecharla. Y, por último, la DM gestacional, que aparece en el período de gestación en una de cada diez embarazadas [2].

Este trabajo se centra en la DM tipo 1. Los enfermos de este tipo de DM necesitan, de por vida, regular su concentración de glucosa en sangre. Para conseguir dicha regulación tienen que medir la cantidad de glucosa presente en la sangre, conocida como glucemia, e inyectarse insulina subcutánea.

En la práctica clínica habitual, la glucemia se puede medir de dos maneras diferentes. Por un lado, se puede determinar de una manera directa, extrayendo la sangre del paciente con un pinchazo. De esta forma se mide la glucosa en sangre, también llamada glucosa capilar. Esta medición se realiza habitualmente cada 8 horas, aunque puede ocurrir que por diversos motivos se realice con más frecuencia.

Y, por otro lado, la glucemia se puede determinar de una manera indirecta midiendo la cantidad de glucosa en los tejidos intersticiales. De esta forma se mide la glucosa intersticial. Esta medición se realiza mediante los Monitores Continuos de Glucosa (MCG), que miden la glucosa de manera autónoma aproximadamente cada 5 minutos.

Tras conocer la concentración de glucosa en sangre de un paciente, se le inyecta insulina con el fin de regularizar dicha concentración. La insulina puede ser suministrada bien manualmente, o bien, mediante un infusor subcutáneo continuo, también llamado bomba de insulina.

Medir la glucemia mediante MCG tiene diversos problemas. En primer lugar, no se mide la glucosa capilar sino una magnitud relacionada como es la glucosa intersticial. La glucosa intersticial presenta dos inconvenientes. Por un lado, el valor obtenido tiene un retraso (lag) de entre diez y quince minutos frente al valor real de la glucosa capilar, es decir presenta una deriva en sus valores frente a los valores de la glucosa capilar. En segundo lugar, existen problemas asociados a las tecnologías utilizadas en el MCG que dan lugar a valores de glucemia incorrectos. Entre ellos:

1. La degeneración con el paso del tiempo de los sensores del MCG.
2. La pérdida de algunas medidas.
3. La existencia de diversos tipos de ruidos durante el proceso de medida.

El fin de este trabajo es estudiar el funcionamiento de los filtros de partículas para reducir los problemas que se enumeraron anteriormente e implementarlo sobre una FPGA. El funcionamiento esperado es que, dada una medición de glucosa de un MCG con cierta degeneración, se puedan

estimar tanto la medida real de la glucosa capilar como la degeneración del MCG. Para ello, se utilizarán los dos tipos de mediciones de la glucosa explicados anteriormente: la medición mediante el pinchazo y la obtenida a través del MCG.

1.2 Problemas de estimación de estados

Un gran número de problemas científicos requieren de una estimación de los estados del sistema. Principalmente cuando estos estados son ocultos, varían a lo largo del tiempo y las mediciones son indirectas y tienen ruidos. El presente trabajo busca estimar la evolución en tiempo discreto del estado de un determinado sistema dinámico. Para ello se utilizan diferentes ecuaciones para modelar la evolución en tiempo discreto del estado del sistema y de las medidas [3].

El estado del sistema se representa mediante un vector de estados. Este vector contiene toda la información relevante requerida para describir el sistema. Por ejemplo, en problemas de seguimiento de objetos (*tracking*), esta información podría estar relacionada con las características cinemáticas del objetivo. Por otro lado, el vector de mediciones representa las observaciones (con cierto ruido) que se encuentran relacionadas con el vector de estados.

Con el fin de analizar el sistema dinámico se necesitan, al menos, dos modelos. En primer lugar, el modelo dinámico del sistema que describa la evolución del estado en el tiempo. En segundo lugar, el modelo de medición que relacione las mediciones, habitualmente con ruido, y el estado. Se asume que ambos modelos están basados en probabilidad. La formulación del espacio de estados probabilístico y el refresco de la información con nuevas mediciones son soportadas perfectamente por la aproximación *bayesiana*.

En este tipo de aproximación se usa una Función de Densidad de Probabilidad (PDF, de sus iniciales en inglés) para la estimación dinámica del estado. Esta PDF se aplica al estado en base a toda la información obtenida, incluyendo el conjunto de mediciones. En principio, una estimación óptima del estado se obtiene mediante esta PDF. En la mayoría de los problemas se necesita una estimación para cada medida obtenida.

Una aproximación al filtrado recursivo supone que los datos recibidos pueden ser procesados secuencialmente. De esta forma, no es necesario almacenar el conjunto completo de datos. Este tipo de filtros consiste principalmente en dos fases: predicción y actualización. La fase de predicción utiliza el modelo dinámico del sistema para predecir el estado. Mientras que, la operación de actualización utiliza la última medida para modificar el estado.

Existen principalmente dos tipos de filtrado que dan solución a estos problemas de estimación de estados:

- Algoritmos óptimos.
- Algoritmos sub-óptimos.

1.2.1 Algoritmos óptimos.

En esta sección se explican los dos principales algoritmos óptimos para la estimación de espacios de estados:

- Los filtros Kalman.
- Los filtros basados en grid.

Los filtros *Kalman* son una de las herramientas más conocidas y utilizadas para la estimación de mediciones de sensores con ruidos. Su nombre lo recibe de *Rudolph E. Kalman*, quien en 1960 publicó un artículo describiendo una solución recursiva al problema del filtrado de datos discretos [4].

El filtro Kalman consiste en un conjunto de ecuaciones matemáticas que implementan un estimador de tipo predictor-corrector. Dicho estimador es óptimo en cuanto a que minimiza la covarianza del error estimado. Desde su introducción, este tipo de filtros han sido utilizados en una gran variedad de aplicaciones. Particularmente ha sido muy utilizado en el área de la navegación asistida.

Por otro lado, los métodos basados en *grid* proveen de una recursión óptima a la densidad filtrada. Esto ocurre si el espacio de estados está discretizado en un número finito de estados [3]. Este tipo de estimación puede ser observado como una evolución de la PDF de la estimación del estado [5].

1.2.2 Algoritmos sub-óptimos

En muchas situaciones los algoritmos óptimos no son aplicables y son necesarias ciertas aproximaciones. En esta sección se presentan tres aproximaciones de filtros *bayesianos* no lineales: el filtro *Kalman* extendido, los métodos aproximados basados en *grid*, y los filtros de partículas.

En primer lugar, el filtro de *Kalman* extendido es la versión no-lineal del algoritmo del filtro de *Kalman*. En este caso, el problema se convierte en lineal utilizando una estimación de la media y la covarianza actuales [6].

En segundo lugar, los métodos aproximados basados en *grid*. Este tipo de filtros son una aplicación de los métodos basados en *grid* [3] en un contexto con un intervalo fijo y suavizado.

Y, por último, los filtros de partículas. Estos algoritmos consisten en aplicar las técnicas de Monte Carlo [7] para aportar una solución al problema de la estimación de estados. La idea subyacente es representar la PDF mediante un número aleatorio de muestras (partículas) con unos pesos asociados y calcular la estimación en base a esas muestras y esos pesos.

1.2.3 Conclusiones

Para la realización de este trabajo se ha decidido utilizar los filtros de partículas.

Se ha tomado esta decisión por tres motivos fundamentales:

- Este tipo de filtros funciona de una manera correcta frente a ruidos no *gaussianos*, que son el tipo de ruido que sufren los datos del MCG.
- Esta aproximación teóricamente produce buenos resultados frente a variables como la degeneración de los sensores.
- Debido a que ya se ha aplicado la solución de un filtro *Kalman* a este problema y resulta interesante la comparativa de ambos proyectos.

1.3 Modelos

Con el fin de aplicar los filtros de partículas primero hay que definir el modelo dinámico (o de proceso) y el modelo de medición. En este trabajo se van a aplicar los modelos propuestos por *Kuure-Kinsey* en su artículo "*A Dual-Rate Kalman Filter for Continuous Glucose Monitoring*" del año 2006 [8]. Las ecuaciones de este método son:

$$X_{k+1} = \Phi X_k + \mathrm{I}^{\omega}\omega_k \quad \textbf{(1.1)}$$

$$Y_k = CX_k + v_k \quad \textbf{(1.2)}$$

La ecuación $X_{k+1} = \Phi X_k + \mathrm{I}^{\omega}\omega_k$ **(1.1)** representa el modelo dinámico del proceso. Mediante esta ecuación se calculan los estados del sistema en el instante k+1, X_{k+1}, en función del estado en el instante k, X_k. La expresión se compone de cuatro elementos:

- La matriz Φ, que relaciona el estado actual con el estado siguiente.
- El estado actual, X_k.
- La matriz I^{ω}, que relaciona cada uno de los estados del vector con un ruido de proceso.
- El vector de ruido asociado a las entradas ω_k (asumido como ruido blanco *Gaussiano*).

La ecuación $Y_k = CX_k + v_k$ **(1.2)**, representa el modelo de medición. En esta ecuación se calcula la medición estimada en el instante k en función del estado en ese instante y un ruido de medición, Y_k. Se compone de tres elementos:

- La matriz C, que relaciona la medición con el estado.
- El estado actual, X_k.
- Y, el vector de ruido asociado a las medidas v_k (asumido como ruido blanco *Gaussiano*).

1.3.1 Modelo dinámico de la glucosa capilar

El modelo consta de cuatro estados, a saber:

- La glucosa en sangre, g_k.
- La velocidad de cambio de la glucosa en sangre, Δg_k.
- La ganancia del sensor, a_k.
- La velocidad de variación de la ganancia del sensor, Δa_k.

La glucosa en sangre se calcula utilizando su ratio de cambio tal y como se ve en la siguiente ecuación:

$$g_{k+1} = g_k + \Delta g_k \quad \textbf{(1.3)}$$

Por otro lado, el ratio de cambio de la glucosa está modelado como una señal estocástica que varía según un ruido $\omega_{g,k}$:

$$\Delta g_{k+1} = \Delta g_k + \omega_{g,k} \quad \textbf{(1.4)}$$

La representación matricial de estas dos ecuaciones es la siguiente:

$$\begin{bmatrix} g_{k+1} \\ \Delta g_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} g_k \\ \Delta g_k \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \omega_{g,k} \quad \textbf{(1.5)}$$

Las mismas ecuaciones se aplican a la ganancia del sensor. Las ecuaciones de la ganancia del sensor y de su velocidad de cambio se representan en notación matricial:

$$\begin{bmatrix} a_{k+1} \\ \Delta a_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a_k \\ \Delta a_k \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \omega_{a,k} \quad \textbf{(1.6)}$$

En definitiva, el modelo dinámico del proceso, que modela los cuatro estados, se representa mediante:

$$\begin{bmatrix} g_{k+1} \\ \Delta g_{k+1} \\ a_{k+1} \\ \Delta a_{k+1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} g_k \\ \Delta g_k \\ a_k \\ \Delta a_k \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \omega_{g,k} \\ \omega_{a,k} \end{bmatrix} \quad \textbf{(1.7)}$$

Se observa que este modelo es una representación de la ecuación $X_{k+1} = \Phi X_k + \mathrm{I}^{\omega}\omega_k$ **(1.1)** donde X_{k+1} y X_k están representadas por las matrices que agrupan los estados de la glucosa y de la ganancia del sensor en los instantes de tiempo $k+1$ y k, respectivamente. Por otro lado, la matriz Φ es la matriz que selecciona qué entradas utilizar para cada una de las estimaciones. Por otro lado, se observan las matrices $\mathrm{I}^{\omega}\ \omega_k$, que muestran qué ruido utilizar para cada una de las estimaciones.

Por último, destacar que el modelo que se utiliza exactamente en este trabajo modifica la ecuación del incremento de la ganancia. A esta ecuación se le añade una constante de degeneración calculada para el problema específico. Esta constante de degeneración proviene de realizar un análisis en el que la ganancia decae exponencialmente con el tiempo, de forma que:

$$a_k = a_0 * e^{-d*k} \quad \textbf{(1.8)}$$

Por lo que, en el siguiente instante de tiempo:

$$a_{k+1} = a_0 * e^{-d*(k+1)} \quad \textbf{(1.9)}$$

De esta forma, a partir de la ecuación $a_{k+1} = a_0 * e^{-d*(k+1)}$ ***(1.9)*** se puede representar a_{k+1} en función de a_k:

$$a_{k+1} = a_0 * e^{-d*(k+1)} = a_0 * e^{-d*k} * e^{-k} = a_k * e^{-d} \quad \textbf{(1.10)}$$

De modo que la degradación en la ganancia del sensor resulta ser e^{-d}. En concreto, se supone que el sensor se degrada a la mitad transcurridos seis días (puesto que se realizan muestras cada 5 minutos, resultarían $5 * 24 * 6 = 720$ muestras en 6 días). Es por ello, que en este caso en concreto la constante de degeneración toma como valor 0,9996.

De esta forma, la ecuación del incremento de la ganancia del sensor, resulta:

$$\Delta g_{k+1} = d * \Delta g_k + \omega_{g,k} \quad \textbf{(1.11)}$$

1.3.2 Modelos de medición

Este modelo se utiliza para calcular la medición estimada en función del estado actual. Como hemos explicado con anterioridad nuestro sistema utiliza dos medidas de la glucosa, una rápida proporcionada por el MCG y otra lenta proporcionada por el pinchazo en el dedo. Por lo tanto, se debe tener dos modelos de medición estimada. El modelo rápido depende tanto de la concentración de glucosa en sangre como de la ganancia del sensor y viene dado por la ecuación:

$$y_{f,k} = \begin{bmatrix} 0.5a_k & 0 & 0.5g_k & 0 \end{bmatrix} \begin{bmatrix} g_k \\ \Delta g_k \\ a_k \\ \Delta a_k \end{bmatrix} + v_{f,k} \quad \textbf{(1.12)}$$

Cabe destacar que este primer modelo es dinámico debido a la presencia de g_k y a_k en la matriz C. Esto significa que el modelo se adapta a la degeneración a lo largo del tiempo.

Por otro lado, el modelo lento sólo depende de la glucosa en sangre y viene dada por la ecuación:

$$y_{s,k} = [1 \quad 0 \quad 0 \quad 0] \begin{bmatrix} g_k \\ \Delta g_k \\ a_k \\ \Delta a_k \end{bmatrix} + v_{s,k} \qquad \textbf{(1.13)}$$

Capítulo 2

2 Filtro de partículas

2.1 Algoritmia del filtro de partículas

Esta sección está dedicada a explicar detalladamente el funcionamiento de los filtros de partículas (FP).

En este filtro se utiliza un conjunto de estados probables del sistema (llamado *grid* de partículas) para solucionar el problema de filtrado. Como todos los filtros *bayesianos*, el filtro de partículas tiene una fase de predicción, en la que se aplican los modelos dinámicos del proceso para que éste evolucione, y una fase de corrección, en la que se utilizan las mediciones obtenidas para corregir las predicciones de la fase anterior. En los FP, además, existe una tercera fase, llamada fase de *remuestreo*, en la que se genera una nueva población de estados probables del sistema. Cada uno de estos estados (también conocido como partícula) cuenta con un peso asociado que indica cómo de buena es la estimación de esa partícula. La etapa de *remuestreo* consiste en obtener un nuevo *grid* de partículas a partir de aquéllas que tengan un mayor peso.

La Figura 2.1 ilustra cómo se relacionan las distintas fases del filtro de partículas. En las siguientes secciones se describe con más detalle cada una de ellas.

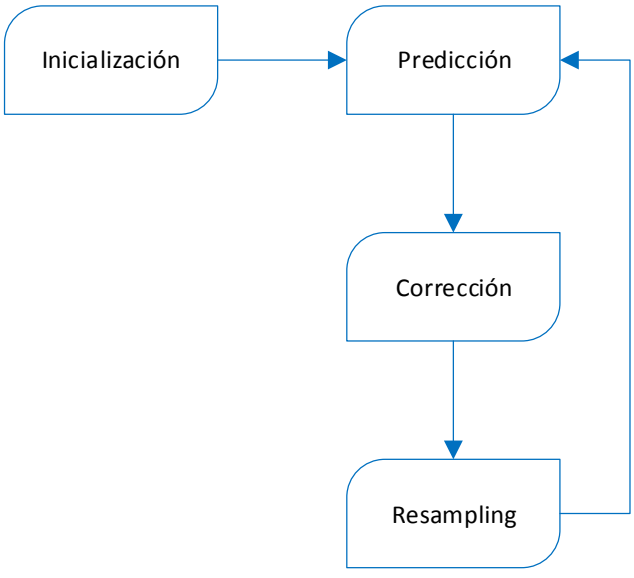


Figura 2.1 – Fases de un filtro de partículas.

2.1.1 Inicialización

En esta fase se genera de forma aleatoria el primer *grid* de partículas. Tal y como se comentó en la sección 1.3, cada partícula es un vector con cuatro estados$[g \quad \Delta g \quad a \quad \Delta a]$, donde:

- g representa la glucosa en sangre del paciente.
- Δg representa la variación de la glucosa en sangre.
- a representa la degradación del sensor.
- Δa representa la variación de la degradación del sensor.

El número de partículas que debe tener el *grid* es uno de los parámetros a estudiar del algoritmo. Una población con un número reducido de partículas puede ser evaluada más rápidamente, pero sus soluciones son menos precisas. Es por eso que se debe encontrar un número de partículas que ofrezca una buena relación entre velocidad de cálculo y precisión. Una vez generada la primera población de partículas es necesario asignar un peso a cada una de ellas. En este primer paso se le da el mismo peso a cada partícula. El valor de este peso es $P_i = \frac{1}{N}$, siendo N el número de partículas.

Una vez terminada la etapa de la inicialización, el algoritmo entra en un bucle (véase la Figura 2.1) en el que se ejecutan las siguientes fases:

1. Predicción.
2. Corrección.
3. *Remuestreo*.

2.1.2 Predicción

En la predicción se aplican las ecuaciones dinámicas del sistema a cada una de las partículas del *grid* para estimar su siguiente estado. A diferencia del filtro *Kalman*¸ en la fase de predicción del filtro de partículas se aplica un determinado ruido aleatorio. Tras la ejecución de esta fase se obtiene una nueva población de partículas que ha evolucionado según los modelos del sistema.

En el caso que se centra este proyecto, y estudiando los valores de g y Δg, las ecuaciones que describen el comportamiento dinámico de la glucosa en sangre son:

$$g_{k+1} = g_k + \Delta g_k \qquad \textbf{(2.1)}$$

$$\Delta g_{k+1} = \Delta g_k + \omega_{g,k} \qquad \textbf{(2.2)}$$

Suponiendo que Q_g es la covarianza del error $\omega_{g,k}$, y que $random_g$ es un número *pseudo*-aleatorio, entonces el error a añadir cuando se aplica el modelo en esta fase es:

$$\omega_{g,k} = Q_g * random_g \qquad \textbf{(2.3)}$$

En cuanto a la ganancia del sensor se utilizan las mismas ecuaciones con una covarianza Q_a:

$$a_{k+1} = a_k + \Delta a_k \qquad \textbf{(2.4)}$$

$$\Delta a_{k+1} = \Delta a_k + \omega_{a,k} \qquad \textbf{(2.5)}$$

$$\omega_{a,k} = Q_a * random_a \qquad \textbf{(2.6)}$$

Tanto Q_g como Q_a no suelen ser datos conocidos a prior. Luego son parámetros a estudiar para el correcto funcionamiento del algoritmo.

Por último, en esta fase se realiza la estimación de la medida aplicando los modelos de medición. Tal y como se comentó en la sección 1.3.2, en este caso se utilizan dos modelos: el rápido, que modela la

medida tomada por el MCG cada cinco minutos, y el lento que modela la medida tomada por el pinchazo cada ocho horas.

Al igual que ocurre con los modelos dinámicos del proceso, a estos dos modelos de medición también se les aplica un ruido aleatorio. Este ruido se define utilizando la covarianza del error de la medición, resultando de la siguiente manera:

$$V_s = R_s * random_s \quad \textbf{(2.7)}$$

$$V_f = R_f * random_f \quad \textbf{(2.8)}$$

De la misma forma que en las ecuaciones dinámicas, los números $random$, son números *pseudo*-aleatorios. Tanto R_f como R_s (covarianzas para el ruido del modelo rápido y del modelo lento respectivamente) son otros dos parámetros a estudiar para el correcto funcionamiento del algoritmo. De esta forma, los modelos de medición resultan:

$$y_s = g + R_s * random_s \quad \textbf{(2.9)}$$

$$y_f = g * a + R_f * random_f \quad \textbf{(2.10)}$$

Por último, destacar, que los números aleatorios, variables *random*, siguen una distribución uniforme en el rango [0,1]. Es decir, que la probabilidad de ocurrencia de cualquier número dentro de este rango es la misma [9].

2.1.3 Corrección

Esta etapa calcula los pesos para cada una de las partículas. Como ya se comentó anteriormente estos pesos indican cómo de buena es una determinada partícula y serán utilizados en la siguiente etapa para seleccionar las mejores partículas.

Se utiliza una función de densidad de probabilidad (PDF, *Probability Density Function*) para relacionar la medida real con las medidas estimadas en la anterior fase y determinar el peso de la partícula. La selección de la PDF es vital para el correcto funcionamiento del algoritmo. Por eso en secciones posteriores se realizará un estudio de diversas funciones con el fin de determinar cuál es la más adecuada.

Una vez se ha calculado el nuevo peso de cada partícula, se procede a la normalización[1] de los pesos.

2.1.4 *Remuestreo*

Por último, en la etapa de *remuestreo* (*resampling)*, se genera un nuevo *grid* de partículas en base a los pesos asignados a cada una de ellas en el paso anterior. Es importante que el número de partículas se mantenga constante durante la evolución del algoritmo. En este nuevo *grid* se almacenan las partículas con "mejor peso". En este caso, la definición de *"mejor peso*" varía según el problema. Es por eso que se aplican diferentes algoritmos para la obtención de las mejores partículas.

Tras la ejecución de esta última fase, se obtiene un nuevo *grid* de partículas que ha evolucionado de acuerdo con la dinámica del sistema y que aproxima la PDF de los estados del sistema. A partir de ella se puede obtener un estimador del estado más probable en el que se encuentra el sistema.

[1] El proceso de normalización consiste en ajustar un grupo de valores para que su suma sea igual a uno.

Después de esta etapa vuelve a aplicarse la fase predicción para que el conjunto de partículas continúe evolucionando.

2.2 Explicación práctica

Esta sección muestra un ejemplo sencillo de funcionamiento de un filtro de partículas. En él se toma una simplificación del problema abordado en este proyecto. En esta simplificación, la estimación de la medida de la glucosa en sangre se corresponde directamente con el valor de la glucosa del conjunto de estados del sistema, quedando así el modelo de medición de la siguiente manera:

$$y_k = [1 \quad 0 \quad 0 \quad 0] \begin{bmatrix} g_k \\ \Delta g_k \\ a_k \\ \Delta a_k \end{bmatrix} \qquad \textbf{(2.11)}$$

En primer lugar, durante la etapa de inicialización se genera un primer *grid* de partículas aleatorio, quedando algo similar a la Figura 2.2.

Figura 2.2 - Fase de inicialización del filtro de partículas.

En esta figura se observa en el eje de las X el tiempo, mientras que, en el eje de las Y, se muestra el nivel de glucosa. En el instante t_0 se observa la primera población de 10 partículas, representadas mediante los círculos azules.

A continuación, se ejecuta la etapa de predicción, en la que se aplican los modelos dinámicos de la glucosa a este grupo de partículas. De esta forma se calculan los valores del siguiente *grid* de partículas. En la Figura 2.3 se muestra la evolución de cada una de las partículas desde el instante de tiempo t_0 al instante de tiempo t_1. Como se puede observar, pese a que las ecuaciones dinámicas sean idénticas para todas las partículas, en unas ocasiones el valor resultante es superior a la inicial, y en otros casos es inferior. Esto se debe al efecto que produce la adición del ruido. Por otro lado, se aplican los modelos de medición, que para esta simplificación se corresponden con el valor de la glucosa calculado con los modelos dinámicos. Destacar que la medición de la glucosa para el instante de tiempo t1 está representada mediante el punto rojo.

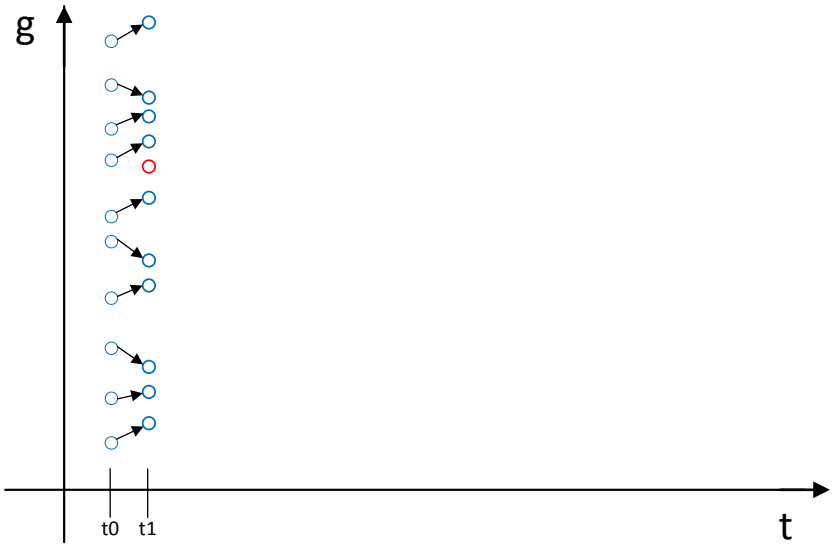


Figura 2.3 - Predicción del filtro de partículas.

Seguidamente se utiliza la etapa de corrección, donde utilizando la estimación de la medida de la glucosa se calcula un peso para cada una de las partículas. Posteriormente se realiza la normalización de estos pesos. Y, por último, se aplica la fase de *remuestreo*, donde se genera el nuevo *grid* de partículas introduciendo aquéllas que tengan mejor peso. En este ejemplo, las mejores partículas son aquellas que están más próximas al punto rojo.

De esta forma, tras la ejecución de este algoritmo N veces, se obtendría algo similar a la Figura 2.4. En esta figura se observa la medida de la glucosa representada mediante la curva roja. El resultado esperado consiste en ver cómo las partículas se pegan a esta curva, es decir, que la estimación de las medidas de la glucosa se asemeja a las medidas reales.

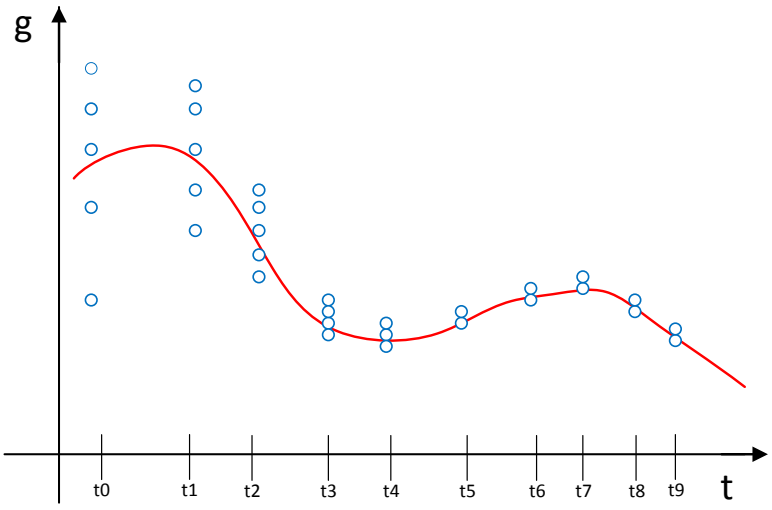


Figura 2.4 - Resultado esperado del filtro de partículas.

Capítulo 3

3 Modelo en *Matlab*

3.1 Introducción

En este capítulo se aborda la implementación de un filtro de partículas para determinar el valor de la glucosa en sangre a partir de las medidas de la glucosa intersticial y de los pinchazos en sangre. El objetivo es realizar una primera prueba de concepto para comprobar el correcto funcionamiento de el algoritmo para este problema.

Por otro lado, también se va a utilizar esta primera aproximación para tomar una serie de decisiones de vital importancia en el algoritmo del filtro de partículas:

1. Fijar la aritmética con la que se trabajará una vez se implemente el algoritmo sobre una FPGA.
2. Decidir qué PDF utilizar.
3. Elegir un método óptimo para la etapa de *remuestreo*.
4. Ajustar los parámetros de los ruidos del modelo dinámico como del modelo de medición (Q_g, Q_a, R_s y R_f).
5. Y, por último, fijar un número óptimo de partículas

Esta prueba de concepto se ha desarrollado utilizando *MATLAB*, ya que proporciona un conjunto de herramientas que facilitan la implementación del algoritmo. En las siguientes secciones se analizan las decisiones sobre los puntos anteriores y se explica de forma detallada el funcionamiento del *script MATLAB*.

3.2 Aritmética de punto fijo

3.2.1 ¿Qué es punto fijo?

El punto fijo es una representación de números reales que cuenta con la ventaja de poder hacer conversiones a y desde números naturales de forma prácticamente inmediata. Como desventaja, el dominio de los valores que puede representar este tipo de aritmética es reducido e impreciso comparado con la notación científica. Sin embargo, a igualdad de *bits*, el punto fijo tiene mayor precisión.

El punto fijo se basa en una representación de números naturales en la que los valores reales han sido multiplicados por un factor antes de ser expresado como número natural. Para recuperar el valor real, deberá dividirse por dicho factor. Si se prefija un factor de 1.000, el valor real de 0,0034 será expresado como $0{,}0034 * 1000 = 3$, por lo que la precisión máxima está determinada directamente por el factor utilizado [10].

3.2.2 Notación *Q*

Dado que el factor de escala puede variar, la "notación Q" se utiliza para especificar el número de dígitos a la derecha del punto. Una notación Q_n indica que el número de *bits* a la derecha del punto es *n*. En general la notación Q no especifica la longitud de la palabra. Cuando es necesario, se utiliza una notación más compleja $Q_{m,n}$ domde m con los *bits* para la parte entera y n los *bits* para la parte decimal. Además, se reserva un *bit* para representar el signo, de esta forma el número de *bits* para esta notación resulta: $N = n + m + 1$. En la Figura 3.1 se muestra la representación de un número binario con notación $Q_{m,n}$ [10].

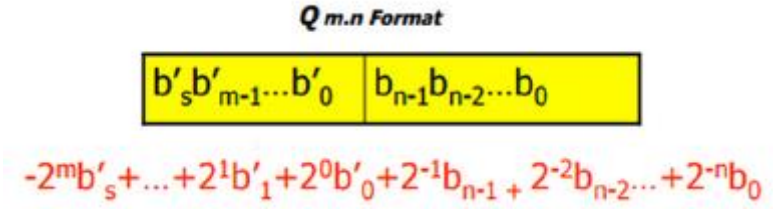


Figura 3.1 - Notación Qn,m.

3.2.3 Operar en punto fijo

Debido a que cada valor se multiplica por una constante, se debe tener en cuenta a la hora de realizar las operaciones. Si se consideran los números reales a y b y f el factor de conversión utilizado, sea $A = a * f$ y $B = b * f$ las operaciones aritméticas básicas son: [10]:

- Suma: $A + B = a * f + b * f = (a + b) * f \rightarrow a + b \sim A + B$
- Multiplicación: $A * B = a * f * b * f = (a * b) * f^2 \rightarrow a * b \sim \frac{(A*B)}{f}$
- División: $\frac{A}{B} = \frac{a*f}{b*f} = \frac{a}{b} \rightarrow \frac{a}{b} \sim \frac{A*f}{B}$

Se debe tener en cuenta que para obtener el valor real o en punto fijo se debe hacer lo siguiente:

- Dada la variable real a, la variable en coma fija A se obtiene como: $A = a * f$.
- Dada la variable en coma fija A, la variable real a se obtiene como: $a = A/f$.

Es interesante remarcar que las operaciones de suma y resta generan resultados de la misma longitud que sus operandos. Sin embargo, no resulta así en las operaciones de multiplicación o división. Por ejemplo, para los valores reales 0,5 (A) y 0,25 (B) en notación Q_{15}, su representación binaria sería 0100000000000000 y 0010000000000000 respectivamente. El resultado esperado (0,125) no es representable con esta notación. Consideremos que:

$$a = \frac{\hat{a}}{2^{15}} \qquad \textbf{(3.1)}$$

$$b = \frac{\hat{b}}{2^{15}} \qquad \textbf{(3.2)}$$

Donde $\hat{a}$ y $\hat{b}$ representan el valor binario de ambos números. De esta forma, el producto de ambos números se representará como:

$$c = a * b = \frac{\hat{a}*\hat{b}}{2^{30}} \qquad \textbf{(3.3)}$$

Así se observa que el resultado de esta operación genera una notación Q_{30}. La norma para la operación de este tipo de aritmética, es que si se operan dos números con notaciones Q_n y Q_m, el resultado será representable con una notación Q_{n+m}.

3.2.2 Aritmética de punto fijo en *Matlab*

Para utilizar la aritmética de punto fijo en *Matlab* es necesario realizar las siguientes acciones [11].

En primer lugar, se debe definir el tipo de números a tratar. La sentencia a ejecutar se muestra en la Figura 3.2. En esta llamada se indica si los números a representar tienen signo, el número de bits completo para representar el número (*WordLength*) y el número de bits a utilizar como parte fraccional (*FractionLenght*).

```
KF_t = numerictype( 'Signed', true, ...
                    'WordLength', numberBits, ...
                    'FractionLength', fractionalBits);
```

Figura 3.2 - Definición de aritmética en Matlab - 1.

A continuación, se define el tipo de aritmética a utilizar (véase la Figura 3.3). En esta función se indican los siguientes campos:

- Método de redondeo. En este caso se ha escogido redondear al valor representable más cercano.
- Acción cuando se excedan los rangos. Se ha escogido saturar. De esta manera, aquéllos datos que no sean representables en la aritmética fijada se saturarán a cero. Es decir, tomarán como valor cero.
- Modo de producto. Se ha indicado que se utilice la precisión indicada en los campos *ProductWordLength* y *ProductFractionLength*.
- Ancho de datos tras una multiplicación (tanto entero como decimal). Se ha decidido que tras una operación de multiplicación el resultado mantenga la misma aritmética.
- Modo de suma. Se ha indicado que se utilice la precisión indicada en los campos *SumWordLength* y *SumFractionLength*.
- Ancho de datos tras una suma (tanto entero como decimal). Se ha decidido que tras una operación de suma el resultado mantenga la misma aritmética.

```
KFArith = fimath('RoundingMethod'    , 'nearest', ...
          'OverflowAction'    , 'Saturate', ...
          'ProductMode'       , 'SpecifyPrecision', ...
          'ProductWordLength', numberBits, ...
          'ProductFractionLength', fractionalBits, ...
          'SumMode', 'SpecifyPrecision', ...
          'SumWordLength', numberBits, ...
          'SumFractionLength', fractionalBits);
```

Figura 3.3 - Definición de aritmética en Matlab - 2.

Y, por último, es necesario indicar que la aritmética utiliza un formato extendido (véase la Figura 3.4). De no ser así, la visualización de los resultados que utilicen esta aritmética aparecerá truncada [12].

```
format LONGG;
```

Figura 3.4 - Definición de formato extendido en Matlab.

3.3 Función de densidad de probabilidad

Como ya se ha comentado en secciones anteriores, es necesario realizar un estudio sobre la función de densidad de probabilidad (PDF) a aplicar a la hora de calcular los nuevos pesos de las partículas. En este caso, se estudiaron tres opciones: la densidad de probabilidad normal, una función de cuantificación (que se explicará más adelante) y el cálculo de la distancia entre la estimación y la realidad.

La función de densidad de probabilidad normal es la más utilizada actualmente en los filtros de partículas [13]. Sigue la siguiente ecuación:

$$P(x) = \frac{1}{\sqrt{2\mu\sigma^2}}^{-\frac{(x-\mu)^2}{2\sigma^2}} \qquad \textbf{(3.4)}$$

La aplicación de esta función al cálculo del peso de una partícula, x_i, da lugar a la siguiente expresión.

$$P_{k+1}(x_i) = P_k(x_i)\frac{1}{\sqrt{2\mu\sigma^2}}^{-\frac{(y_i-\hat{y_i})^2}{2\sigma^2}} \qquad \textbf{(3.5)}$$

Siendo $P_{k+1}(x_i)$ el peso final para la partícula x_i, $P_k(x_i)$ el peso inicial, σ la varianza de la función de densidad de probabilidad. El símbolo y_i representa la medida de la glucosa y el símbolo $\hat{y_i}$ la estimación de la medida de la glucosa.

Por otro lado, también suele utilizarse una función de cuantificación, que es una función más sencilla de implementar. Esta función se modela de la siguiente manera:

$$P_{k+1}(x_i) = \begin{cases} 1 & si\ y_k \leq \hat{y}_k < y_k + \delta \\ 0 & en\ caso\ contrario \end{cases} \qquad \textbf{(3.6)}$$

En esta función el símbolo δ indica un error a tener en cuenta al comparar el valor de $\hat{y}_k$ e y_k.

Por último, se estudió utilizar la distancia entre la medida estimada ($\hat{y}_k$) y la medida real (y_k) tal y como se muestra en la siguiente ecuación:

$$P_{k+1}(x_i) = \frac{1}{abs(\hat{y}_k - y_k)} \qquad \textbf{(3.7)}$$

En este trabajo, se ha decidido utilizar la distancia entre la estimación y la realidad, en primer lugar, debido a que la implementación en hardware de una función de densidad normal es muy costosa. Y, en segundo lugar, a que una función de cuantificación de este tipo únicamente proporciona o el mejor peso (1) o el peor (0), pero no valores intermedios.

3.4 Método de *remuestreo*

A la hora de implementar la etapa de *remuestreo* es muy importante fijar un método para seleccionar las partículas con mejor peso. En este trabajo se han estudiado dos métodos: la ruleta y el torneo. Ambos métodos implementan un proceso estocástico en el que es más probable seleccionar aquellas partículas con un mayor peso.

El método de la ruleta consiste en asignar a cada partícula una serie de valores proporcionales a sus pesos y seleccionar de forma aleatoria entre toda la población en el instante k qué partículas formarán parte de la población en el instante k+1. La selección es aleatoria, pero la probabilidad de selección es directamente proporcional al peso de la partícula. Para ilustrar su funcionamiento, se va a considerar las siguientes partículas (con sus pesos normalizados correspondientes):

- Partícula 1 → peso 0,15.
- Partícula 2 → peso 0,25.
- Partícula 3 → peso 0,1.
- Partícula 4 → peso 0,3.
- Partícula 5 → peso 0,2.

En la Figura 3.5 se observa la repartición de estos pesos en una ruleta virtual:

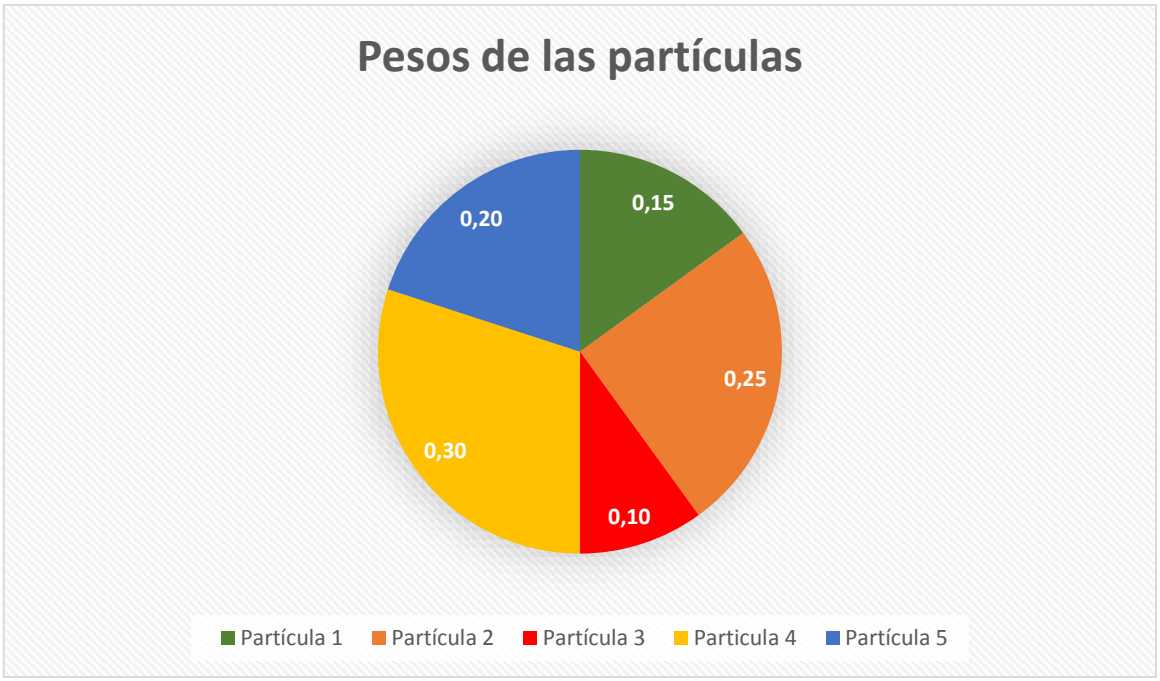


Figura 3.5 - Método de la ruleta. Asignación de secciones de la ruleta para los pesos del ejemplo.

De esta manera, se genera un número aleatorio entre el 0 y el 1, y dependiendo de en qué sección de la ruleta de la Figura 3.5 caiga, se escoge una partícula u otra. Se continúan generando números aleatorios hasta que se obtiene un *grid* del mismo tamaño que el *grid* de partida.

El método de torneo consiste en generar dos números aleatorios y escoger las partículas representadas por esos números. A continuación, se comparan los pesos de ambas partículas y se añade a la población aquella que tiene mayor peso. Al igual que en el método anterior, se itera el proceso hasta que se consigue un nuevo *grid* de partículas del mismo tamaño que el inicial.

Finalmente se decidió utilizar el método del torneo por su sencillez de implementación en hardware.

3.5 Obtención del número de partículas óptimo

Un número de partículas adecuado es fundamental para el correcto funcionamiento del algoritmo. En esta sección presento el método que he ideado para determinar el número de partículas. Lo que se

busca es un número mínimo de partículas a partir del cual la estimación de los estados ocultos no mejore al aumentar dicho número. Se busca un número mínimo ya que la complejidad computacional aumenta con el número de partículas.

El método que se estableció, similar al error cuadrático medio (age de las siglas en inglés del error medio en la glucosa), se representa mediante:

$$age = \sqrt{(\frac{\sum_{i=0}^{N} \frac{\sum_{j=0}^{N_P} \hat{y}_{i,j}}{N_P}}{N} - y)^2} \qquad \textbf{(3.8)}$$

Tal y como se observa en la fórmula anterior, esta medida se basa en la distancia entre la estimación de la glucosa ($\hat{y}$) y la glucosa real (y). Primero de todo, destacar que el valor N indica el número de mediciones tomadas. Esta distancia se eleva al cuadrado con el fin de considerar de la misma manera las distancias positivas que las negativas. Posteriormente se realiza la suma de todas las iteraciones y se calcula la raíz cuadrada de dicha suma con el fin de contrarrestar el cuadrado anterior. Por último, el número obtenido se divide entre el número de iteraciones para obtener la diferencia final.

3.6 *Scripts* para el filtro de partículas

En esta sección se explican los tres *scripts* que se han desarrollado para el cálculo del algoritmo del filtro de partículas. En primer lugar, el *script particle_filter.m*, que se encarga de realizar el cálculo del algoritmo y del error cuadrático medio (ECM, que se explicará más adelante). En segundo lugar, el *script run_particle_filter.m*, que sencillamente ejecuta el *script* anterior con el fin de comprobar si las estimaciones de los estados ocultos son correctas. Y, en tercer lugar, el *script max_particles.m*, que se encarga de analizar la evolución del ECM según aumenta el número de partículas.

En todos los scripts se va a utilizar la aritmética de punto fijo descrita en la sección 3.2.2. Para la definición de esta aritmética, es necesario proporcionar el ancho de bits tanto de la parte entera como de la parte fraccional. Debido a que el segundo *script* (*run_particle_filter.m*) y el tercer *script* (*max_particles.m*) utilizan números muy diferentes (se explica posteriormente), ambos anchos se proporcionan al primer *script* desde el resto.

En las siguientes subsecciones se explican con más detalles tanto estos *scripts* como los resultados obtenidos.

3.6.1 Script *particle_filter.m*

Este *script* implementa la función *particle_filter*. Esta función recibe cuatro entradas: (i) el ancho de la parte entera de la representación en punto fijo, *numberBits*; (ii) el ancho de la parte fraccional, *fractionalBits*; (iii) el número de partículas a utilizar por el filtro, *MAX_PARTICLES*; (iv) un *booleano* para indicar si se quiere o no mostrar las figuras, *showFigures*. La función devuelve un valor age que representa el error medio de la glucosa, tal y como se explicó en la sección 3.5, y muestra por pantalla la estimación de los estados ocultos del sistema (glucosa y ganancia del sensor).

En este *script*, en primer lugar, se definen los parámetros configurables del filtro, tal y como se observa en la Figura 3.6.

```
Qg_b10 = 0.10;
Qa_b10 = 0;
Rs_b10 = 0;
Rf_b10 = 0;

Qg = fi( Qg_b10, KF_t, KFArith );                % Glucose increase noise
Qa = fi( Qa_b10, KF_t, KFArith );                 % Gain increase noise
Rs = fi( Rs_b10, KF_t, KFArith );                 % Slow measurement noise
Rf = fi( Rf_b10, KF_t, KFArith );                 % Fast measurement noise
```

Figura 3.6 - Parámetros configurables en Matlab.

Cabe destacar que estos valores se inicializan en base 10 para posteriormente aplicarles la aritmética de punto fijo (utilizando la función *fi*).

Tras una serie de ejecuciones del *script* se decidió únicamente aplicar ruido en el modelo dinámico de la glucosa. Estas decisiones se explican más adelante.

Tras definir la aritmética y los parámetros configurables, se generan unas medidas sintéticas de entrada para comprobar la validez del filtro. Estos valores corresponderán a las medidas reales en el algoritmo. Para generar estas medidas se necesita definir cómo es la glucosa en sangre y cómo es la ganancia del sensor. Esto se debe a que las medidas con las que trabaja el filtro son, por un lado, la rápida que es el producto de la ganancia del sensor y la glucosa en sangre, y, la lenta, que es directamente el valor de la glucosa en sangre. (véase la Figura 3.7). Esto se realiza con el fin de mostrar los estados ocultos reales y los estados ocultos calculados por el filtro. De esta manera, se puede verificar el correcto funcionamiento del algoritmo.

```
%% 3. Inputs generation (glucose & detector gain)
% Gain function
T0=2800;              % Decrease time
t = 1:5785;           % Sampling time
gain_b10=exp(-t/T0);     % Decreasing exponential function
gain = fi( gain_b10, KF_t, KFArith );

% Glucose function
a_b10=100; %amplit
p_b10= 0;
f_b10=0.01;
x_b10=0:1:5784;

BG_b10=(a_b10*tan(sin(f_b10*x_b10+p_b10))+a_b10*cos(f_b10*x_b10+p_b10)).*exp(-t/T0);
BG = fi( BG_b10, KF_t, KFArith );

% Real measurement
patientRawData=BG.*gain;
```

Figura 3.7 - Estados ocultos y medida real.

Para este ejemplo la medida real de la glucosa en sangre viene representada en la Figura 3.15. Por otro lado, la ganancia del sensor sigue una exponencial decreciente (véase la Figura 3.16). Mientras que la glucosa sigue una función basada en senos, cosenos y arco tangentes (como puede observarse

en la Figura 3.18). Por último, cabe destacar que la medida real se obtiene con la multiplicación de la ganancia y de la glucosa. De esta manera se consigue que las entradas sintéticas se degraden en función de la ganancia del sensor.

A continuación, se genera el primer *grid* de partículas de forma aleatoria y se le indica el peso para cada partícula:

```
%% 5. Pool particles creation
particles_b10 = [70 .* rand(MAX_PARTICLES, 1) + 70, (20).*rand(MAX_PARTICLES,1) + -10, ones(MAX_PARTICLES, 1), zeros(MAX_PARTICLES,1)];
weights_b10   = 1/MAX_PARTICLES * ones(MAX_PARTICLES,1);

particles = fi( particles_b10, KF_t, KFArith );
weights = fi( weights_b10, KF_t, KFArith );
```

Figura 3.8 - Fase de inicialización en Matlab.

El tipo de valores que se asigna a cada atributo de las partículas:

- Para la glucosa, se asignan valores aleatorios entre 70 y 140. Esto se debe a que el rango de valores de la glucosa se encuentra entre estos dos números [14].
- Para el incremento de la glucosa, se asignan valores aleatorios entre 10 y 20. De esta forma se indica que la glucosa puede variar en este rango de valores de un intervalo de tiempo al siguiente.
- Para la ganancia del sensor, se asigna siempre uno como valor inicial.
- Y, por último, para el incremento de la ganancia del sensor, se aplica siempre cero. Con estos dos últimos valores, se consigue una ganancia constante siempre a uno, pero al aplicar los modelos se irá degenerando este valor con el fin de representar una exponencial decreciente.

Tras la fase de inicialización, se entra en un bucle que itera sobre el resto de fases del algoritmo. La siguiente etapa a ejecutar será la etapa de predicción, que implementa los modelos tal y como se muestra más adelante:

```
% Prediction phase
% Apply glucose models
particles(:,1) = particles(:,1) + particles(:,2);
particles(:,2) = particles(:,2) + 2*Qg * fi( rand(MAX_PARTICLES, 1), KF_t, KFArith ) - Qg;

% Apply gain models
particles(:,3) = fi( 0.9996, KF_t, KFArith ) * particles(:,3) + particles(:,4);
particles(:,4) = particles(:,4) - Qa;

% Slow model
if ( mod( idx,96 ) == 0 )
    % Apply model in order to estimate the real measurement
    ye = particles(:,1) + Rf * rand( MAX_PARTICLES,1 );
% Fast model
else
    % Apply model in order to estimate the real measurement
    ye = particles(:,1) .* particles(:,3) + Rs * rand( MAX_PARTICLES,1 );
end
```

Figura 3.9 - Fase de predicción en Matlab.

Cabe destacar que el modelo de medición lenta se aplica cada 8 horas (el pinchazo directo del paciente), mientras que el modelo de medición rápida se aplica cada 5 minutos (la medición del MCG).

Posteriormente se aplica la fase de corrección, donde se calcula el nuevo peso para cada partícula en base a la cercanía de su estimación con la realidad (véase la Figura 3.10). Tras este cálculo se normalizan los pesos de todas las partículas.

```
% Correction phase
% PDF
if ( abs(ye - glucose.Data(idx)) ~= 0 )
    weights = 1./abs(ye - glucose.Data(idx));
end

% Normalizing weights
sumWeights = sum(weights);

if ( sumWeights ~= 0 )
    weights = divide( KF_t, weights, sumWeights );
end
```

Figura 3.10 - Fase de corrección en Matlab.

A continuación, se aplica la fase de *remuestreo*, en la que se genera un nuevo *grid* de partículas usando el método de torneo (sección 3.4), tal y como se observa en la Figura 3.11.

```
% Fase de resampling
tournament_b10 = randi(MAX_PARTICLES, MAX_PARTICLES, 2);
tournament = fi( tournament_b10, KF_t, KFArith );

for k = 1:MAX_PARTICLES
    if weights(tournament(k,1)) > weights(tournament(k,2))
        new_particles(k,:) = particles(tournament(k,1),:);
    else
        new_particles(k,:) = particles(tournament(k,2),:);
    end
end
```

Figura 3.11 - Fase de remuestreo en Matlab.

Los datos de los estados ocultos obtenidos en cada iteración se almacenan para su posterior análisis (véase la Figura 3.12). Tal y como se observa, en primer lugar, se modifican las partículas (variable *particles*) a utilizar para calcular la siguiente iteración con las nuevas partículas calculadas (variable *new_particles*).

```
% Collect data
particles                = new_particles;
glucose_particles(:,idx) = particles(:,1);
gain_particles(:,idx)    = particles(:,3);
```

Figura 3.12 - Recolección de cálculos en Matlab.

Y, por último, se muestran, o no, las figuras según se indica con el parámetro *showFigures* y se calcula el valor de error age, tal y como se observa en la Figura 3.13.

```
%% 6. Results representation
if showFigures

    figure(2);
    plot(mean(glucose_particles), '.');
    xlabel('Sample');
    ylabel('Glucose'); % left y-axis
    title('Estimated glucose measurement');

    figure(3);
    plot(mean(gain_particles), '.');
    xlabel('Sample');
    ylabel('Gain'); % left y-axis
    title('Estimated gain measurement');

    figure(4);
    plot(BG);
    xlabel('Time (minutes)');
    ylabel('BG (mg/dl)'); % left y-axis
    grid on;

    figure(5);
    plot(gain);
    xlabel('Time (minutes)');
    grid on;
end

age =  sqrt( sum( ( ( mean( glucose_particles ) - BG ) .^ 2 ) ) ) / 5784;
```

Figura 3.13 - Figuras y cálculo de ECM.

3.6.2 Script *run_particle_filter.m*

Este *script* llama a la función *particle_filter*, descrita en la sección anterior, con los siguientes parámetros (véase la Figura 3.14):

- 27 bits de anchura para los números de representación de punto fijo.
- 15 bits de anchura para la parte decimal (lo que deja $27 - 15 = 12$ bits para la parte entera.
- 10 partículas.
- Y, por último, se indica que se quiere que se muestren las figuras generadas.

```
% Se establecen los parametros que definen la aritmetica a emplear
numberBits      = 27;
fractionalBits = 15;

age = particle_filter(numberBits, fractionalBits, 10, true);

clear all;
clc;
```

Figura 3.14 - Script run_particle_filter.m

Tras la ejecución de este *script*, se obtienen las siguientes figuras:

- La Figura 3.15 representa la medida real de la glucosa.
- La entrada sintética para la ganancia del sensor se muestra en la Figura 3.16. Se observa que la Figura 3.17, donde se muestra la estimación de la ganancia del detector, se asemeja en gran medida a los valores reales.
- Y, por último, ocurre lo mismo con la glucosa sintética mostrada en la Figura 3.18 frente a la estimación de la misma, representada en la Figura 3.19.

De esta forma, se observa de manera visual que las estimaciones tanto de la ganancia del sensor como de la glucosa se asemejan a la realidad.

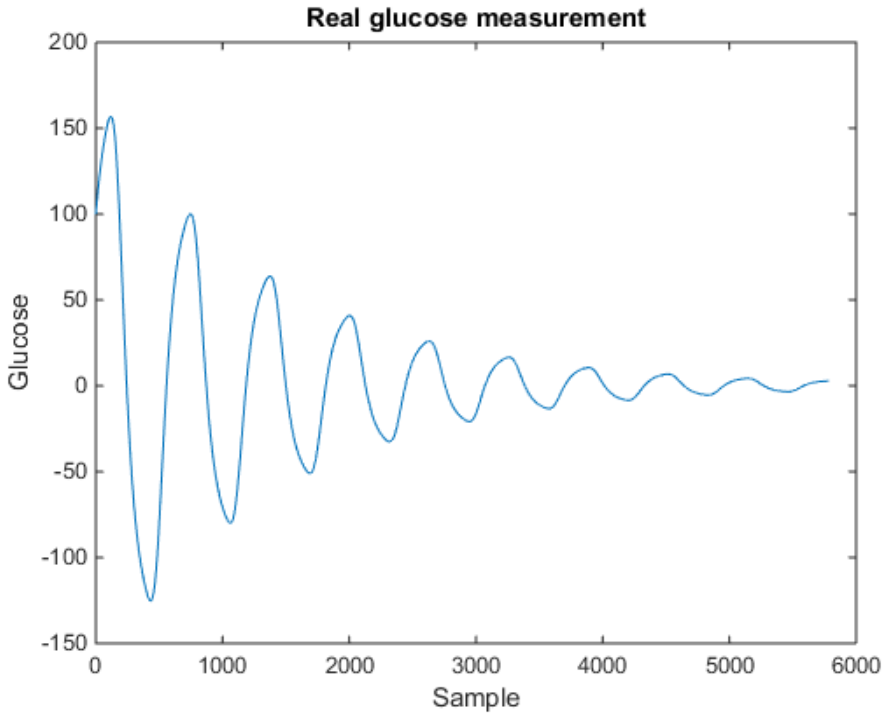


Figura 3.15 - Medida real.

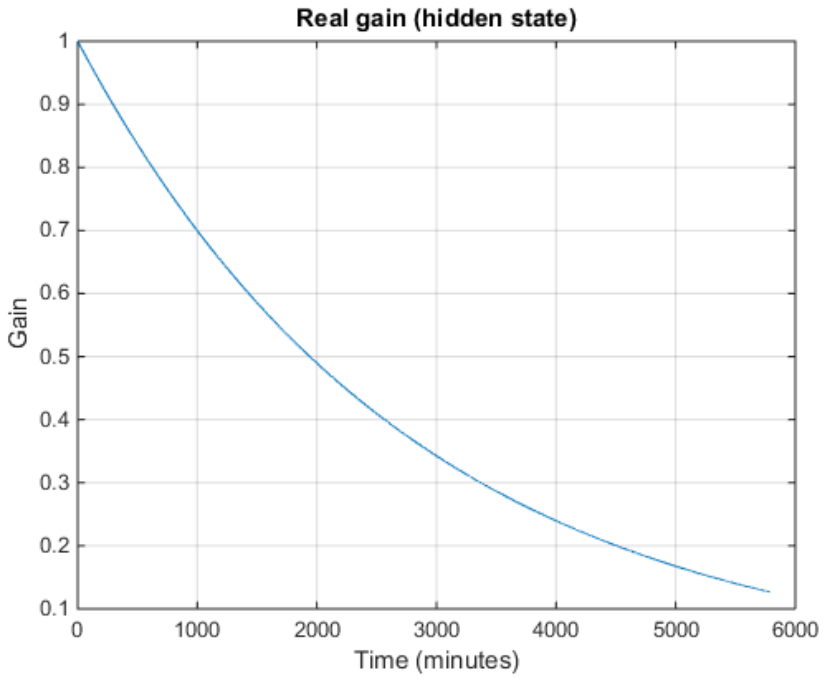


Figura 3.16 - Ganancia del sensor (estado oculto).

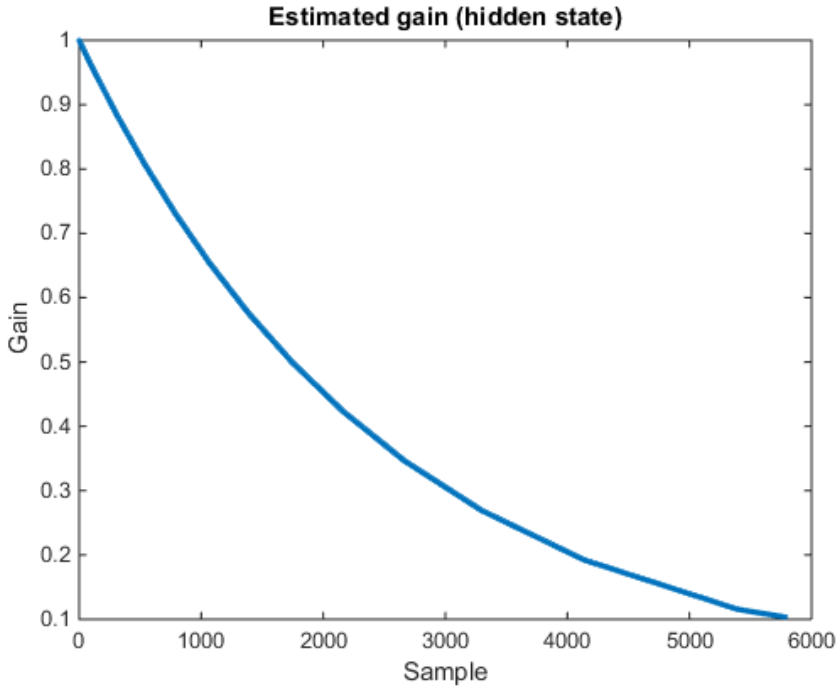


Figura 3.17 - Ganancia del sensor estimada.

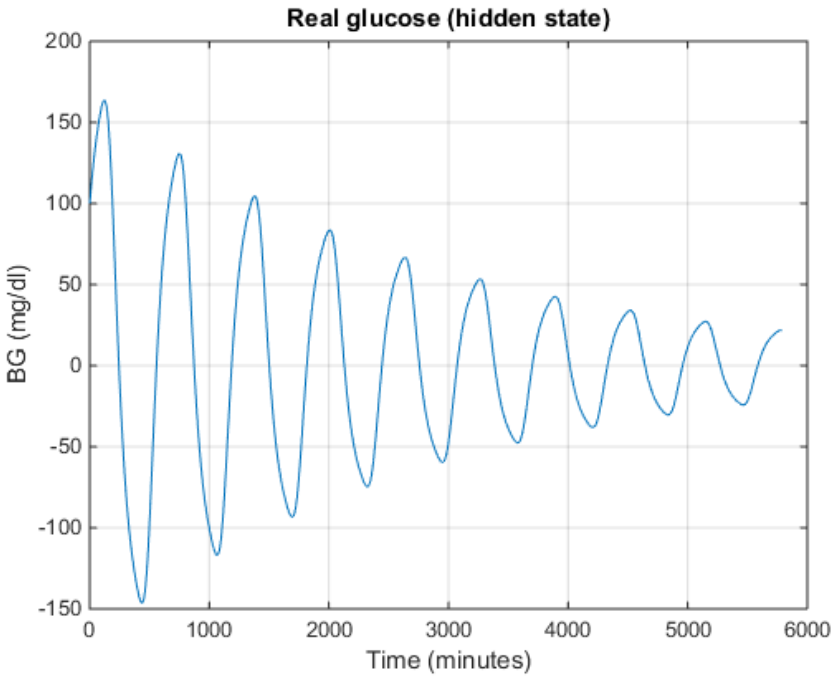


Figura 3.18 - Glucosa en sangre (estado oculto).

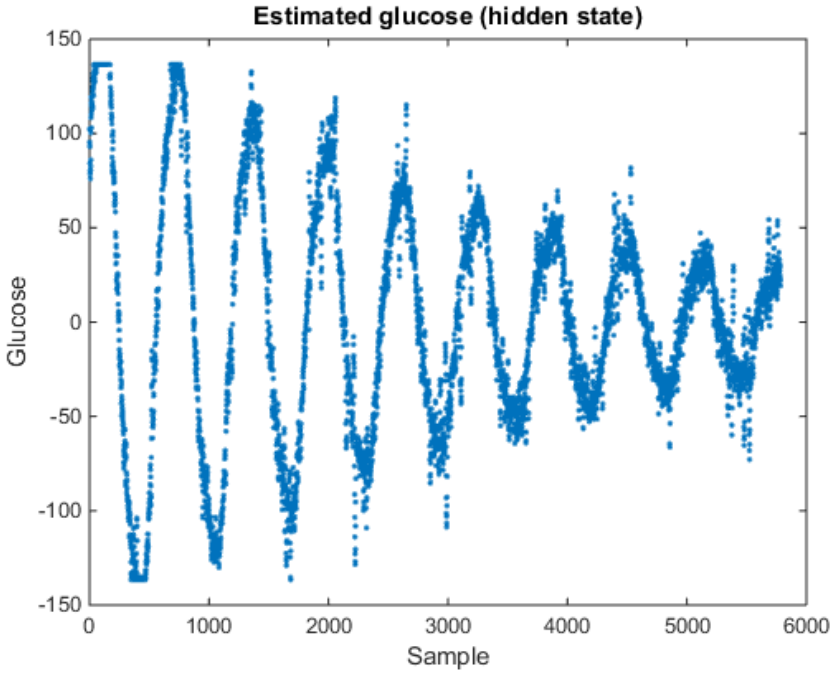


Figura 3.19 - Glucosa estimada.

3.6.3 *Script max_particles.m*

Este *script* ejecuta L veces la función *particle_filter* aumentando el número de partículas con el fin de analizar el error medio de la glucosa (age) para cada número de partículas. Tal y como se observa en la Figura 3.20, el age calculado se guarda en una tabla (*age_t*), que posteriormente se muestra en una imagen.

```
format LONGG;

% Se establecen los parametros que definen la aritmetica a emplear
numberBits      = 27;
fractionalBits = 15;

for N_PARTICLES = 1:25
    age = particle_filter(numberBits, fractionalBits, N_PARTICLES, false);
    age_t(N_PARTICLES) = age;
end

figure(1);
plot(age_t, '-');
ylabel('Average Glucose Error');
xlabel('#PARTICLES');
title('age');

clear all;
```

Figura 3.20 - Script max_particles.m.

La Figura 3.21 se obtiene al ejecutar el script para un número de partículas desde 1 hasta 25. En esta figura se observa cómo evoluciona el error age (representado en el eje Y) según evoluciona el número de partículas (representado en el eje X). Los valores para números de partículas bajos (entre 1 y 4) se encuentran entre los valores 1.000 y 7.000. Esto se debe a que el filtro no funciona bien para un número de partículas tan bajo. Sin embargo, a partir de 5 partículas, se observa que el error age pasa a valer casi cero.

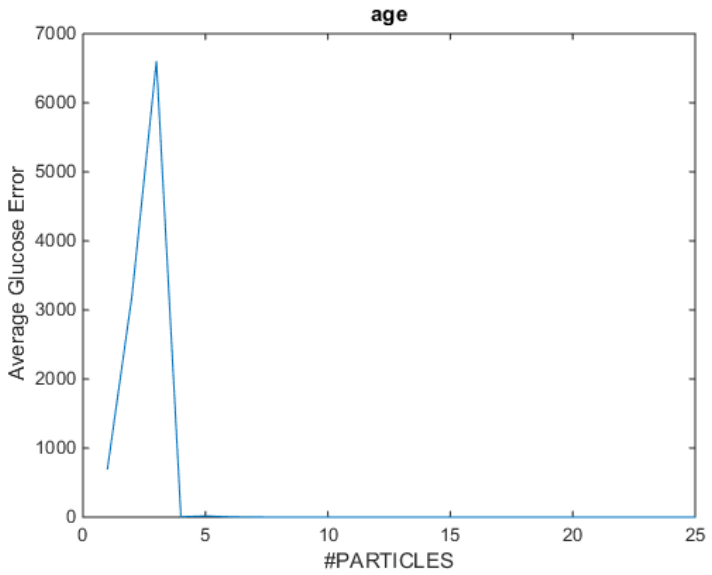


Figura 3.21 – Error *age para un intervalo de 1 a 25 partículas.*

Es por esto, que se realizó una segunda ejecución con un número de partículas desde 9 hasta 15, mostrando el error age como un punto para observar con mayor claridad la evolución de este error para estos casos en concreto. Los resultados de esta segunda ejecución se observan en la Figura 3.22.

En esta figura se observa que, por un lado, el age es cero entre los valores 1 y 8. Esto es debido a que el *array* que guarda los ECM no tiene valor para estos índices, con lo cual los inicializa con el valor cero. Es decir, estos valores no son valores reales. Y, por otro lado, a partir de la partícula 8, el age no llega a valer uno, lo cual significa que el error del filtro para estos números de partículas es casi ínfimo.

Figura 3.22 – Error *age para un intervalo de 8 a 15 partículas.*

Tras este estudio, se decidió utilizar 10 partículas. Con este valor se consigue, por un lado, mantener estable el age y, por otro lado, reducir el coste computacional del algoritmo por ser un número pequeño.

Capítulo 4

4 Diseño del filtro de partículas

4.1 Introducción

En este capítulo se explica el diseño *VHDL* del filtro de partículas utilizando la herramienta *HLDesigner*.

En primer lugar, se explica el uso del paquete *fixed_pkg* desarrollado por el IEEE para el uso de aritmética en punto fijo en VHDL, puesto que se utilza en todo el desarrollo del diseño.

A continuación, se explican dos librerías de uso común por todos los módulos del diseño: la librería *common* y la librería *random*.

Por último, se explica el diseño propiamente dicho. Se seguirá un orden *top-bottom*, es decir, se explican primero los módulos más generales para ofrecer una visión global del diseño antes de entrar en los detalles del mismo.

4.2 Paquete *fixed_pkg*

Como se ha visto en la sección 3.2 un número con parte fraccionaria puede representarse como un entero. De esta forma, en VHDL se puede utilizar directamente aritmética de enteros. Pero, tal y como también se explicó con anterioridad, a las operaciones de multiplicación y división hay que aplicarles ciertas operaciones de corrección para que los resultados continúen teniendo la misma base implícita. Aunque está aritmética se puede implementar directamente, la tarea es algo tediosa. Esta es la razón de usar el paquete *fixed_pkg* de IEEE puesto que facilita enormemente la tarea. En este paquete se encuentran una gran cantidad de operaciones con aritmética de punto fijo (tales como la suma, la resta, la división, la multiplicación, etc.) y una gran variedad de funciones (como por ejemplo funciones de *cast*, de redondeo y saturación, funciones que devuelven el valor absoluto, etc.).

Tal y como se explicó en la sección 3.2, en este tipo de aritmética hay que indicar cuántos bits de parte entera y de parte decimal se van a utilizar. En este paquete esta información se proporciona al definir el rango de las señales. Si queremos una aritmética con E bits de parte entera y D bits de parte decimal, el rango se define como (E-1 *downto* D). En la Tabla 4.1 se puede ver un ejemplo de declaración de una señal llamada data utilizando el paquete *fixed_pkg*. Esta señal es de punto fijo con signo y la parte entera utiliza 11 bits y la parte decimal 10

```
signal data :   sfixed( 10 downto -10 );
```

Tabla 4.1 - Declaración de una señal con aritmética de punto fijo utilizando el paquete fixed_pkg.

En el diseño, se han utilizado, por un lado, las operaciones de suma, resta, división y multiplicación. Estas operaciones se implementan mediante los operadores “+”, “-“, “*”, “/”. También es interesante mencionar que todas estas operaciones son sintetizables. En la Tabla 4.2 se observa una operación de suma entre dos señales. En el ejemplo se observa que la operación suma está incluida dentro de una función *resize*, que se explica más adelante. En este caso su uso es obligatorio si queremos mantener

el tamaño del resultado, puesto que el tamaño de la suma tiene por defecto un bit más que los operandos.

Por otro lado, se han utilizado las siguientes funciones:

- *resize*: dado una señal o un puerto representado en esta notación, se le indica qué anchura de bits pasa a tener. Esta función se utiliza después de casi todas las operaciones. En el caso de este proyecto, se desea que todas las operaciones obtengan como resultado una salida del mismo tipo que las entradas. Esto se consigue llamando a esta función con cada operación (tal y como se observa en la Tabla 4.2).
- *to_sfixed / to_ufixed*: dado un número entero realiza un *cast* al tipo indicado.
- *abs*: dado un número en aritmética de punto fijo devuelve su valor absoluto.

```
signal a     :   sfixed( 10 downto -10 );
signal b     :   sfixed( 10 downto -10 );
signal r     :   sfixed( 10 downto -10 );

r <= resize( a + b, r'high, r'low );
```

Tabla 4.2 - Operación de suma utilizando el paquete fixed_pkg.

4.3 Definiciones

En esta sección se proporcionan ciertas definiciones de utilidad para la posterior explicación del diseño en VHDL:

- Bloque embebido se corresponde con un proceso VHDL y no es instanciable.
- Un componente es un bloque instanciable que se corresponde con un programa VHDL completo. A su vez, un componente puede estar compuesto por una serie bloques embebidos y otros componentes.
- Un proceso *for generate* es un bloque embebido en el que se instancian ciertos componentes u otros bloques embebidos un número conocido de veces.

4.4 Librería *common*

En esta librería se incluyen todos aquellos módulos o paquetes que serán utilizados por el resto de librerías. De esta forma se evita la replicación de diseño. A continuación, se explican más en detalle estos módulos y paquetes.

4.4.1 Paquete *definitions*

El paquete *definitions* contiene la definición de contantes (tal y como se muestra Figura 4.1) y tipos de datos usados en el diseño. En primer lugar, se asigna valor a las anchuras de los tipos de datos (se definirán tres tipos de datos que se explicarán más adelante). Cabe destacar que en secciones anteriores se ha utilizado un único tipo de datos. Esto se debe a que en el diseño VHDL es necesario ajustar las anchuras de los datos por separado con el fin de no ocupar más espacio en la placa que el necesario.

1. El tamaño de los datos que representan las partículas (glucosa y ganancia del sensor) están definidos por las constantes *c_data_integer_lenght* y *c_data_decimal_lenght*.
2. El tamaño de los pesos está definido por las constantes *c_weight_data_integer_lenght* y *c_weight_data_decimal_lenght*. El tamaño es diferente porque los pesos están normalizados

y nunca superarán a 1, por lo que es necesaria una mayor anchura de *bits* para la parte decimal.

3. El tamaño de los ruidos aleatorios está definido por las constantes *c_rnd_noise_integer_leght* y *c_rnd_noise_decimal_lenght*. Al igual que ocurre con los pesos, estos ruidos serán pequeños, por lo cual necesitan una mayor anchura de *bits* para la parte decimal.

En segundo lugar, se definen dos constantes necesarias para el filtro de partículas:

1. La constante *c_particle_dimensions*, que representa las dimensiones de las partículas. Esto se refiere a los tipos de partículas que se manejan en este proyecto: $[g \quad \Delta g \quad a \quad \Delta a]$.
2. La constante *c_max_particles* que toma el valor obtenido en el *script* de *Matlab* explicado en el capítulo anterior.

En tercer lugar, se añadieron unas constantes para los *delays* que se encuentran en el diseño. Tal y como se explicará más adelante, a los cálculos les acompañará una señal binaria indicando si el dato es válido o no. Esta señal se registra para introducir retardos en su camino y así sincronizar la salida de los módulos correspondientes con esta señal de validación. Es por eso que se añaden estas constantes para indicar cuántos ciclos retrasar esta señal. Estas constantes son las que aparecen en la sección *Delay constans* en la Figura 4.1.

Y, por último, se define una constante que representa el valor cero como un vector de *bits* de una anchura igual al número máximo de partículas (*c_max_particles*).

```
-- Data type lenght constants
constant c_data_integer_lenght          : natural := 10;
constant c_data_decimal_lenght          : natural := 15;

-- Weight type lenght constants
constant c_weight_data_integer_lenght   : natural := 10;
constant c_weight_data_decimal_lenght   : natural := 20;

-- Random noise type lenght constants
constant c_rnd_noise_integer_lenght     : natural := 2;
constant c_rnd_noise_decimal_lenght     : natural := 10;
constant c_std_rnd_width                : natural  := c_rnd_noise_integer_lenght + c_rnd_noise_decimal_lenght;

-- Particles constants
constant c_particle_dimensions          : natural := 4;
constant c_max_particles                : natural := 10;

-- Delay constants
-- Prediction phase
constant c_en_glucose_model_d           : integer := 2;
constant c_en_gain_model_d              : integer := 3;
constant c_particles_model_d            : integer := 5;
constant c_estimation_delay             : integer := 2;

-- Other constants
constant C_ZERO                         : std_logic_vector( c_max_particles-1 downto 0 ) := ( others => '0' );
```

Figura 4.1 - Definición de constantes del paquete definitions.

Por otro lado, en este paquete, se definen los tipos de datos que se utilizan en el diseño. La Figura 4.2, muestra, en primer lugar, la definición de una serie de subtipos:

- El subtipo *t_data*, que está representado como un número en notación de punto fijo con signo con las anchuras definidas previamente.
- El subtipo *t_weight_data*, que está representado como un número en notación de punto fijo con signo con las anchuras definidas previamente.
- Y el subtipo *t_rnd_noise*, que está representado como un número en notación de punto fijo con signo con las anchuras definidas previamente.

Posteriormente se definen los tipos de datos en base a los subtipos definidos anteriormente. Todos estos tipos son *arrays* del número de partículas indicado previamente.

```
--------------------------------------------------------------------------------
-- Data types
--------------------------------------------------------------------------------
subtype t_data is sfixed(c_data_integer_lenght-1 downto -c_data_decimal_lenght);
subtype t_weight_data is sfixed(c_weight_data_integer_lenght-1 downto -c_weight_data_decimal_lenght);
subtype t_rnd_noise is sfixed(c_rnd_noise_integer_lenght-1 downto -c_rnd_noise_decimal_lenght);

type t_weights is array (0 to c_max_particles-1) of t_weight_data;

type t_glucose is array (0 to c_max_particles-1) of t_data;
type t_delay_tmp_glucose is array (0 to c_max_particles-1) of t_glucose;

type t_ye is array (0 to c_max_particles-1) of t_data;
type t_particle is array (0 to c_particle_dimensions-1) of t_data;

type t_particles is array (0 to c_max_particles-1) of t_particle;
type t_delay_tmp_particles is array (0 to c_max_particles-1) of t_particles;
```

Figura 4.2 - Definición de tipos del paquete definitions.

Y, por último, se definen los parámetros del filtro (Qg, Qa, Qs, Qf y la constante que modela la ganancia). Estas constantes se muestran en la Figura 4.3. Cabe destacar que los valores de estas constantes son los calculados en el *script* de *Matlab*.

```
-- Configurable filter parameters
constant c_Qg                : t_data := to_sfixed(0.10, c_data_integer_lenght-1, -c_data_decimal_lenght);
constant c_Qa                : t_data := to_sfixed(0, c_data_integer_lenght-1, -c_data_decimal_lenght);
constant c_Rs                : t_data := to_sfixed(0, c_data_integer_lenght-1, -c_data_decimal_lenght);
constant c_Rf                : t_data := to_sfixed(0, c_data_integer_lenght-1, -c_data_decimal_lenght);
constant c_gain_model_value  : t_data := to_sfixed(0.9996, c_data_integer_lenght-1, -c_data_decimal_lenght);
```

Figura 4.3 - Parámetros del filtro.

4.4.2 Operadores

Esta librería también contiene componentes que implementan las operaciones que se usan con los tipos *t_data* y *t_weight_data*. Debido a que estos tipos están representados con aritmética de punto fijo se utiliza el paquete *fixed_pkg* de *VHDL*. De esta forma la implementación de estos módulos consiste en utilizar los operadores de este paquete:

- Suma (operador '+').
- Resta (operador '−').
- Multiplicación (operador '∗').
- División (operador '/').

En la Figura 4.4 se observa el símbolo para el sumador de tipo *t_data* (todos los operadores tienen un símbolo parecido). Tal y como se observa tiene dos entradas (los operandos a utilizar) y una salida (el resultado de la operación). Tal y como se explicó en la sección 3.2, con el fin de mantener el tamaño de los datos constante tras estas operaciones, se utiliza la función *resize* del paquete *fixed_pkg* (véase la Figura 4.5).

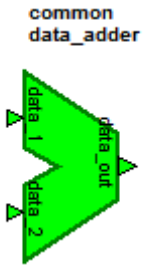


Figura 4.4 - Sumador de tipo t_data.

```
data_out <= resize( data_1 + data_2, data_1'high, data_1'low );
```

Figura 4.5 - Operador de suma en VHDL.

4.4.3 Registros

Otros módulos que se encuentran en esta librería son los registros. Estos registros están diseñados para los tipos *t_data*, *std_logic*, *t_glucose*, *t_particles*, *t_weight_data*, *t_weight* y *t_ye* (teniendo así un registro para cada uno de estos tipos). En la Figura 4.6 se observa el símbolo de uno de estos registros. Estos registros se activan por flanco positivo de reloj y tienen una señal de *reset* síncrono que se activa a nivel bajo.

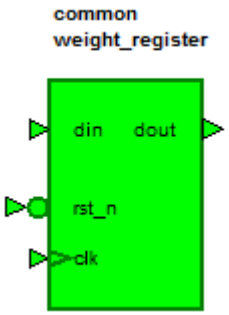


Figura 4.6 – Registro de tipo t_weight.

4.4.4 *Delays*

Por último, esta librería cuenta con una serie de *delays*, que retrasan la señal de entrada un número de ciclos de reloj indicado mediante un genérico (véase la Figura 4.7). Estos módulos han sido diseñados para los tipos *t_glucose*, *t_particles* y *std_logic*.

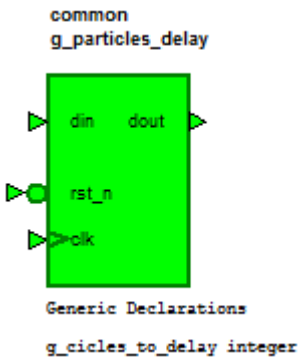


Figura 4.7 – Delay del tipo t_particles.

4.5 Librería *random*

Esta librería cuenta con una serie de componentes para la generación de números *pseudo*-aleatorios. A lo largo de esta sección se da una visión *top-bottom* de los mismos.

Su implementación se ha realizado de acuerdo con [15] y [16].

4.5.1 Componente *signed_pseudo_random*

Esta componente genera un número *pseudo*-aleatorio con un bit de signo. El número generado consta de un bit para el signo y de *g_rnd_width*-1 (siendo *g_rnd_width* un genérico). En la Tabla 4.3 se muestra la interfaz de este módulo.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst	*std_logic*	entrada	Señal de *reset* activa a alta.
out	*std_logic_vector(g_rnd_width-1 downto 0)*	salida	Número *pseudo*-aleatorio con signo generado.

Tabla 4.3 - Interfaz del módulo signed_pseudo_random.

En la Figura 4.8 se observa que este módulo consta de tres componentes. Dos de ellos (*pseudo_rnd_gen*) se encargan de:

- Generar un bit de forma pseudo-aleatoria para el bit de signo (*i_random_sign*).
- Generar el conjunto de bits de forma *pseudo*-aleatoria que representan el valor (*i_random_number*).

Y, el proceso *p_store* que se encarga de concatenar en el puerto de salida el bit de signo y el grupo de bits que representan al número.

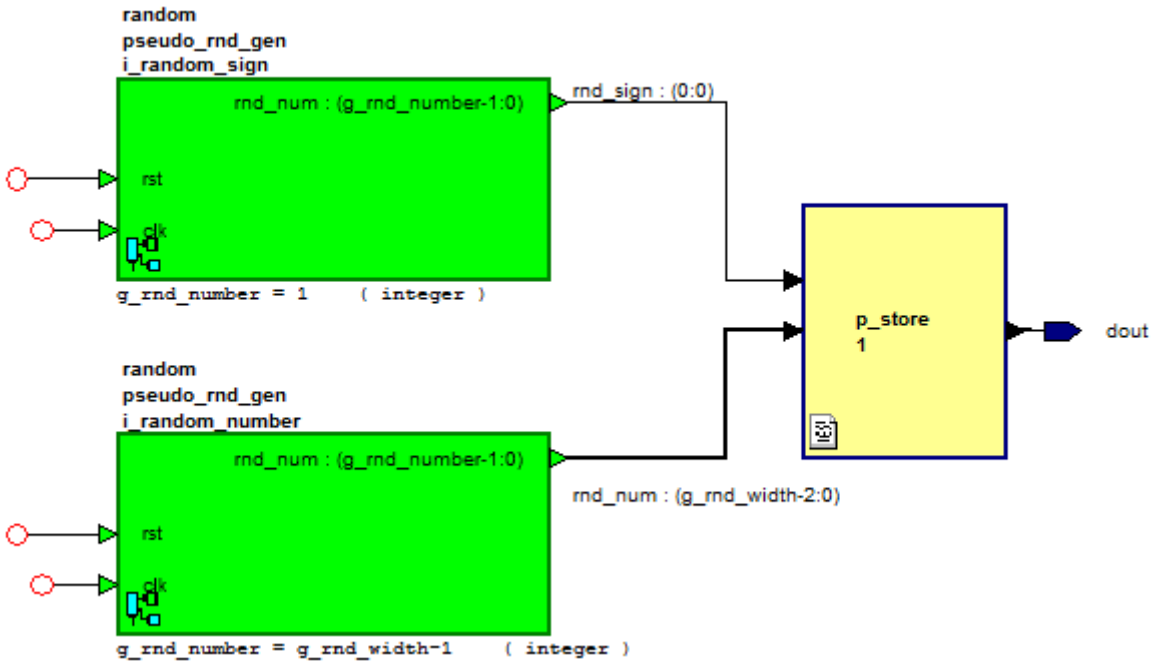


Figura 4.8 - Módulo signed_pseudo_random.

4.5.2 Módulo *pseudo_rnd_gen*

Este módulo se encarga de generar un número *pseudo*-aleatorio con la anchura indicada en el genérico *g_noise_width*. La interfaz del módulo se muestra en la Tabla 4.4 .

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst	*std_logic*	entrada	Señal de *reset* activa a alta.
out	*std_logic_vector(g_rnd_width-1 downto 0)*	salida	Número *pseudo*-aleatorio generado.

Tabla 4.4 - Interfaz del módulo pseudo_rnd_gen.

Este módulo cuenta con un *for generate* que instancia a la entidad *pseudo_random* tantas veces como se indique con el genérico *g_rnd_number*. Por otro lado, también consta de un proceso que genera una señal *std_logic* con valor uno, de esta forma se le envía una señal de capacitación a los módulos *pseudo_random*.

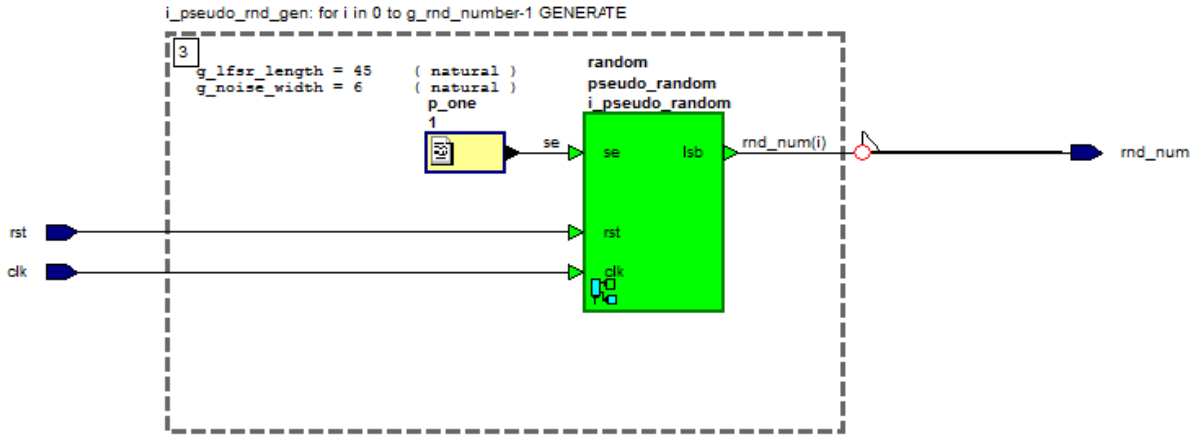


Figura 4.9 - Módulo pseudo_rnd_gen.

4.5.3 Módulo *pseudo_random*

La entidad *pseudo_random* genera un *bit* de manera *pseudo*-aleatoria. La interfaz de este módulo se muestra en la Tabla 4.5.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst	*std_logic*	entrada	Señal de *reset* activa a alta.
se	*std_logic*	entrada	Capacitación del módulo *lfsr_galois*.
lsb	*std_logic*	salida	Bit *pseudo*-aleatorio generado.

Tabla 4.5 - Interfaz del módulo pseudo_random.

Tal y como se observa en la Figura 4.10 en este módulo se instancian dos componentes y dos bloques embebidos. En primer lugar, el proceso *ctrl_fsm*, modela una máquina de estados que se encarga de generar dos señales de capacitación para el resto de componentes (véase la Figura 4.11). Esta máquina de estados cuenta con tres estados, en cada uno de ellos se les da valor a las señales *sipo_en* y *lfsr_ld* de la siguiente manera:

- *init_st:* las señales *sipo_en* y *lfsr_ld* toman valor cero. En el siguiente ciclo de reloj se avanza al estado *load_seed*.
- *load_seed*: la señal *sipo_en* toma el valor uno. Durante 44 ciclos de reloj el sistema se mantiene en este estado. Transcurridos dichos ciclos de reloj, se avanza al estado *end_st*.
- *end_st*: la señal *sipo_en* toma el valor cero y la señal *lfsr_ld* toma el valor uno. En el siguiente ciclo de reloj se avanza al estado *init_st*.

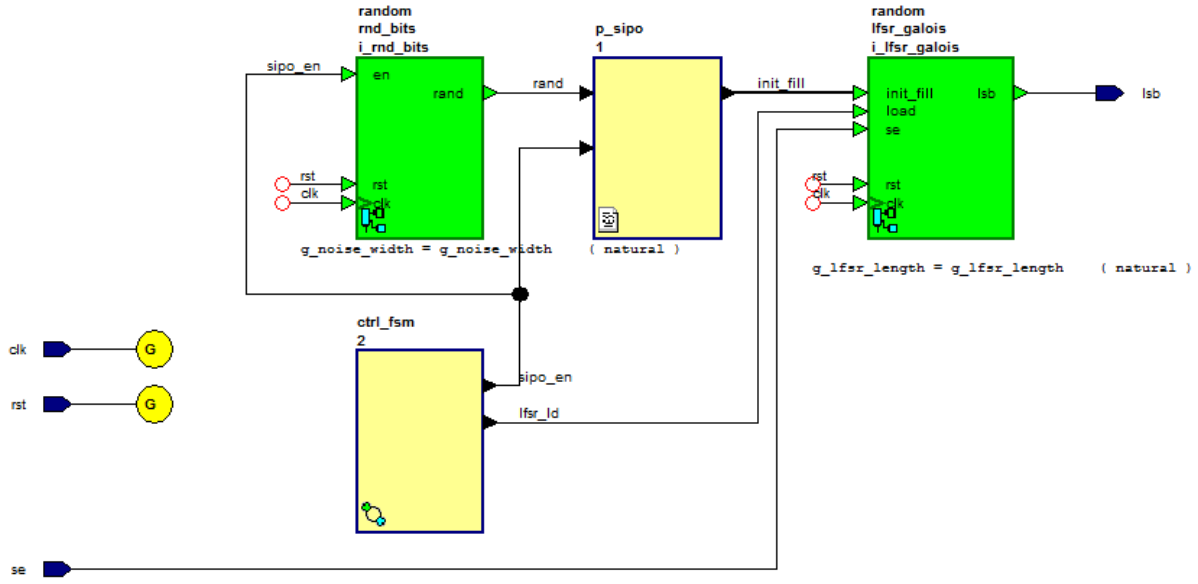


Figura 4.10 - Módulo pseudo_random.

En segundo lugar, el componente *rnd_bits* genera un *bit pseudo*-aleatorio siempre y cuando la señal *sipo_en* generada por la máquina de estados explicada antes tome valor uno. El *bit* aleatorio generado por la entidad *rnd_bit* pasa a un registro de desplazamiento modelado por el proceso *p_sipo* (véase la Figura 4.12). Este proceso recibe una entrada de un *bit* y genera una salida de tantos *bits pseudo*-aleatorios como se indique en el genérico *g_noise_width*.

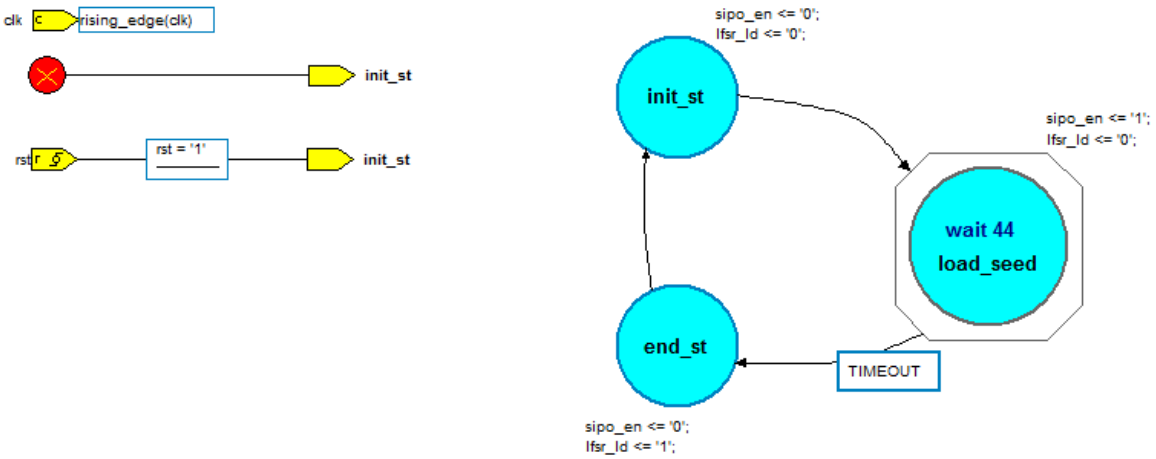


Figura 4.11 - Máquina de estados del módulo pseudo_random.

Y, por último, el componente *lfsr_galois*, genera un *bit pseudo*-aleatorio utilizando una serie de operaciones lógicas (tal y como se explicará más adelante) sobre la señal *init_fill*.

```
p_sipo :  process (clk)
begin
  if rising_edge(clk) then
    if rst = '1' then
      init_fill <= (others => '0');
    elsif sipo_en = '1' then
      init_fill <= init_fill(init_fill'left-1 downto 0) & rand;
    end if;
  end if;
end process p_sipo;
```

Figura 4.12 - Proceso p_sipo.

4.5.4 Módulo *rnd_bits*

Tal y como se ha comentado anteriormente, este módulo genera un *bit* de manera *pseudo*-aleatoria. La interfaz de esta entidad se observa en la siguiente tabla:

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst	*std_logic*	entrada	Señal de *reset* activa a alta.
en	*std_logic*	entrada	Capacitación del módulo.
rand	*std_logic*	salida	Bit *pseudo*-aleatorio generado.

Tabla 4.6 - Interfaz del módulo rnd_bits.

Este módulo consta de un proceso *for generate* en el que se instancia el componente *osc* (véase la Figura 4.13). El módulo *osc* genera números *pseudo*-aleatorios. Ha sido implementado de acuerdo a [17]. Se ha diseñado un bloque IP (de sus siglas en inglés *Intellectual Property*) con un número parametrizable de registros de desplazamiento realimentados (de sus siglas en inglés LFSR), que son inicializados con un generador de números aleatorios (TRNG) basado en el muestreo del oscilador en anillo descrito en [18]. El número y el ancho de los LFSR es configurable, en síntesis, y para este caso, deben tomar los valores 10 y 45 respectivamente. A continuación, se realiza una reducción del vector de *bits* generado por estas instancias utilizando una operación lógica *xor*. De esta forma se genera un único *bit* como el resultado de realizar una operación *xor bit* a *bit* para la señal *rnd*. Por último, se registra este *bit* y se proporciona como puerto de salida.

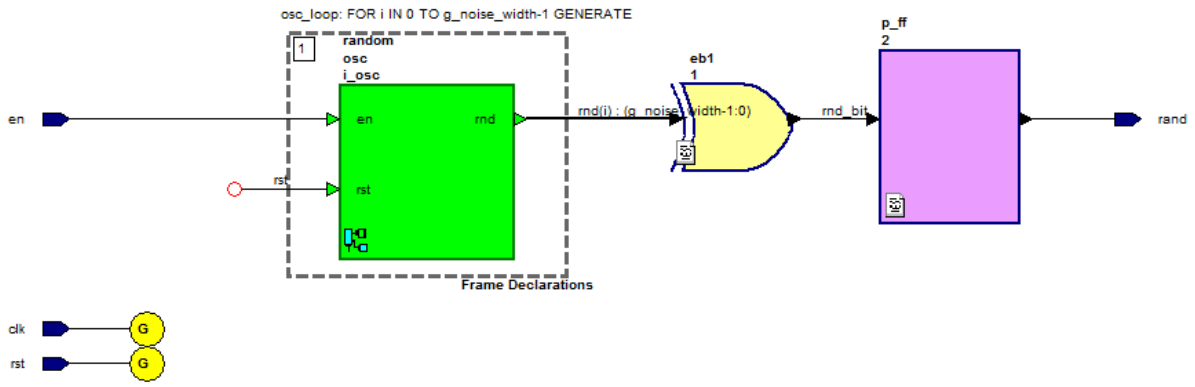


Figura 4.13 - Módulo rnd_bits.

4.5.5 Módulo *lfsr_galois*

Por último, el módulo *lfsr_galois* implementa un registro de desplazamiento con retroalimentación lineal (*linear feedback shift regiter*). Este módulo devuelve un único *bit* generado tal y como se muestra en la Figura 4.14 [19]. La interfaz de este módulo se observa en la Tabla 4.7.

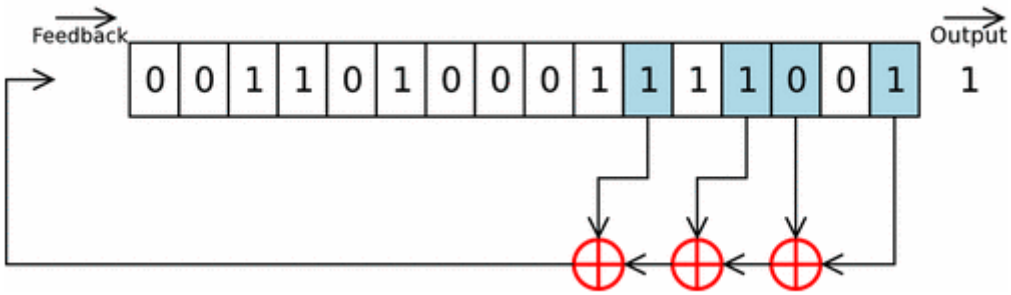


Figura 4.14 - Registro lfsr.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst	*std_logic*	entrada	Señal de *reset* activa a alta.
init_fill	*std_logic_vector(g_lfsr_length-1 downto 0)*	entrada	Valor de inicialización.
load	*std_logic*	entrada	Señal para cargar el valor de *init_fill*.
se	*std_logic*	entrada	Capacitación del módulo.
lsb	*std_logic*	salida	Bit *pseudo*-aleatorio generado.

Tabla 4.7 - Interfaz del módulo lfsr_galois.

Este módulo cuenta con tres procesos, tal y como se observa en la Figura 4.15. En primer lugar, el proceso *p_reg* selecciona el valor a cargar en la señal *r_iq* de la siguiente manera:

- Si la señal de *reset* está activa, se carga todo a ceros.
- Si el puerto *load* toma valor uno, se carga en *r_iq* el valor del puerto *init_fill*.
- Si el puerto *load* toma valor cero, y el puerto *se* toma valor uno, se carga el valor de *nr_iq*.
- En cualquier otro caso, la señal *r_iq* mantiene su valor.

En segundo lugar, el módulo *p_feedback*, modifica la señal *nr_iq* aplicando una serie de operaciones lógicas (*and* y *xor*) sobre la señal *r_iq*. Y, por último, el proceso *p_output* carga el bit menos significativo de la señal *r_iq* sobre el puerto *lsb*.

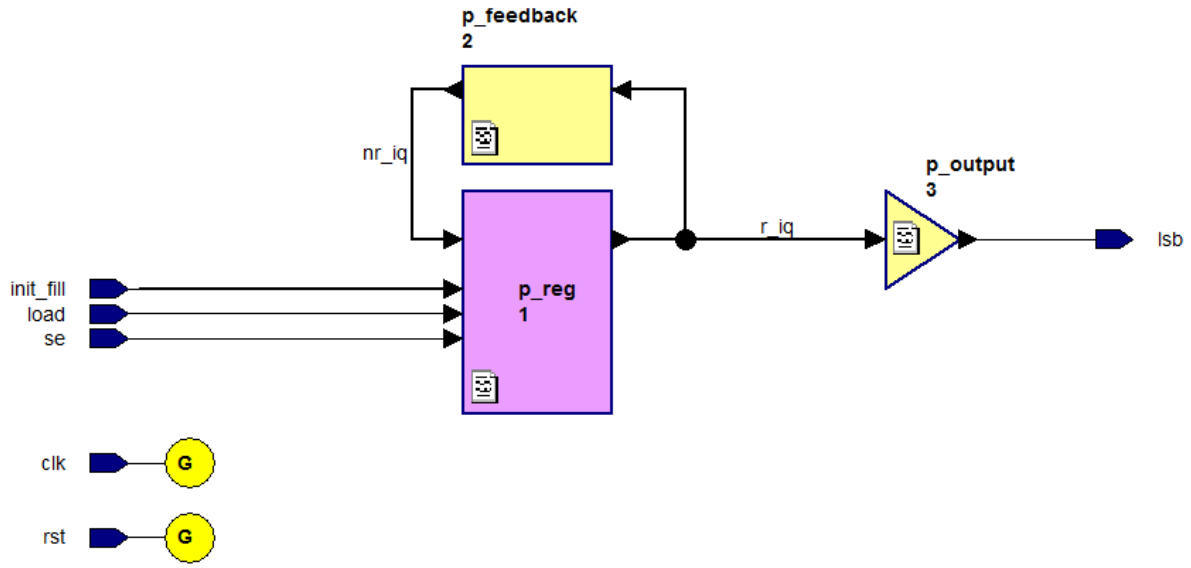


Figura 4.15 - Módulo lfsr_galois.

4.6 Librería particle_filter

En esta librería se encuentra el *top module* (el módulo *particle_filter*). En la tabla Tabla 4.8 se presenta la interfaz de este módulo.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activa a baja.
real_measurement	*t_data*	entrada	Medida real de la glucosa en sangre (o bien a través del pinchazo, o bien a través del MCG).
fast_slow	*std_logic*	entrada	Señal que indica si la medición real es una medición rápida (1) o lenta (0).
particles_from_tb	*t_particles*	entrada	*Grid* de partículas inyectado a través del *testbench* (primera población).
req_from_tb	*std_logic*	entrada	Señal que indica que el *grid* de partículas es válido y que viajará en sincronía con él.
done	*std_logic*	salida	Señal que indica que el nuevo *set* de partículas es válido.
new_particles	*t_particles*	salida	Nuevo *grid* de partículas generado por una iteración del filtro de partículas.

Tabla 4.8 – Interfaz del filtro de partículas.

Tal y como se observa en la Figura 4.16, el *top level* consta de cuatro componentes. Por un lado, se instancian las etapas del algoritmo del filtro de partículas (predicción, corrección y *remuestreo*). Mientras que, por otro, se instancia el módulo *particle_RAM*, que conserva el *grid* de partículas con el que se trabaja durante una iteración. En secciones posteriores se explica con más detalle cada uno de estos módulos.

Durante todo el diseño, los cálculos de las partículas y los pesos estarán acompañados de una señal que indicará su validez. Esto es así para únicamente tener en cuenta los datos válidos e ignorar los

datos inválidos. Esto se debe a que hasta que no se terminen los cálculos del algoritmo, los datos generados no se consideran válidos.

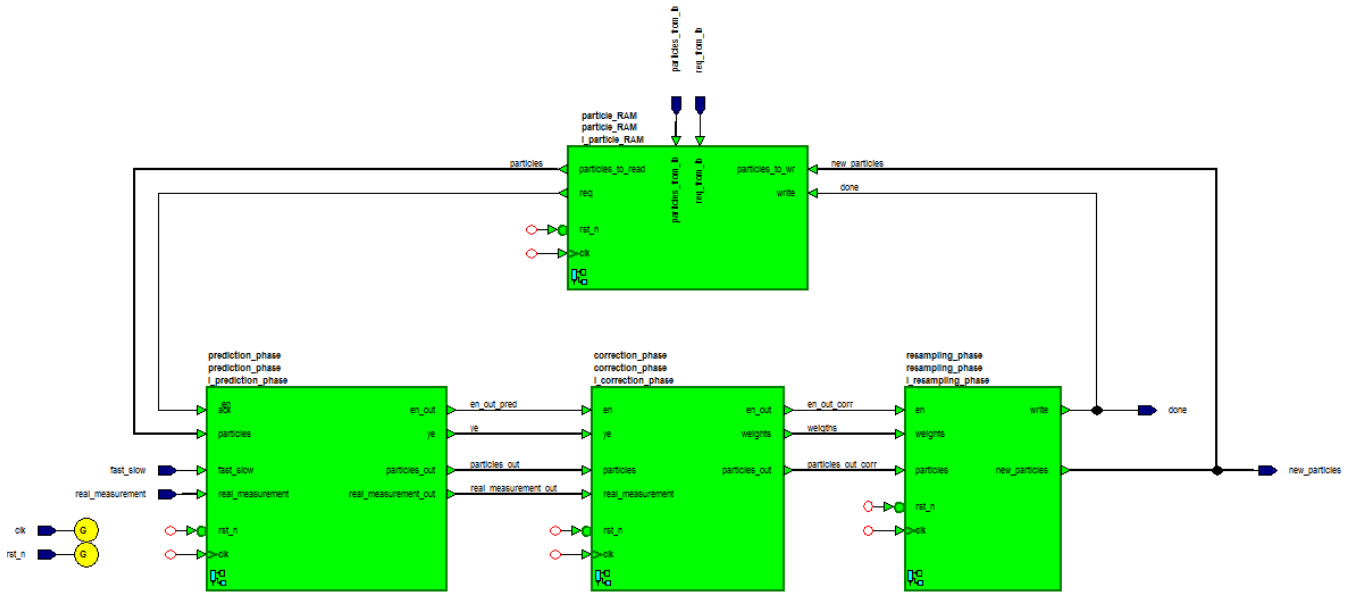


Figura 4.16 - Módulo particle_filter.

4.7 Librería *prediction_phase*

En esta librería se encuentra el módulo *prediction_phase*, que se encarga de realizar la etapa de predicción del filtro de partículas. Es decir, se encarga de aplicar los modelos dinámicos de proceso y los modelos de medición al *grid* de partículas. En la Tabla 4.9 se observa la interfaz de este módulo.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activo a baja.
ack	*std_logic*	entrada	Señal que indica que las partículas de entrada son válidas.
particles	*t_particles*	entrada	*Grid* de partículas sobre el que aplicar los modelos.
real_measurement	*t_data*	entrada	Medida real de la glucosa en sangre (o bien a través del pinchazo, o bien a través del MCG).
fast_slow	*std_logic*	entrada	Señal que indica si la medición real es una medición rápida (1) o lenta (0).
en_out	*std_logic*	salida	Señal que indica que el nuevo *set* de partículas es válido.
particles_out	*t_particles*	salida	Nuevo *grid* de partículas con los modelos aplicados.
ye	*t_ye*	salida	Medición estimada por los modelos de medición.
real_measurement_out	*t_glucose*	salida	Medida real de la glucosa retrasada N ciclos.

Tabla 4.9 - Interfaz del módulo prediction_phase.

Tal y como puede observarse en la Figura 4.17, este módulo se compone de cuatro sub-módulos. El módulo *prediction_phase* se encarga de dos tareas. En primer lugar, aplica los modelos dinámicos de proceso al *grid* de partículas, consiguiendo de esta manera un nuevo *grid* con los nuevos datos estimados de glucosa y de la ganancia del detector. El módulo que realiza estos cálculos se llama *particles_model* (que será explicado más adelante). Es necesario añadir cierto retardo a las señales *real_*measurement y *fast_*slow para sincronizarlas con la generación de las nuevas partículas. Esto se consigue mediante los módulos *g_glucose_*delay y *g_std_*delay explicados en la sección 4.4.4. A lo largo de todo el diseño se utilizarán estos módulos de retardo para sincronizar las señales con los datos calculados[2].

Y, en segundo lugar, el módulo *prediction_phase* se encarga de aplicar los modelos de medición (lento o rápido según indique la señal *fast_slow*) con el fin de obtener la medición estimada de glucosa. De esta tarea se ocupa el módulo *estimation* (que también será explicado posteriormente). Cabe destacar que, aparte de generar la estimación, también produce a su salida el nuevo *grid* de partículas y la medición real. Esto se debe a que ambos datos serán necesarios para etapas posteriores.

[2] En posteriores secciones no se volverá a comentar dichos módulos para no oscurecer la explicación del diseño del algoritmo.

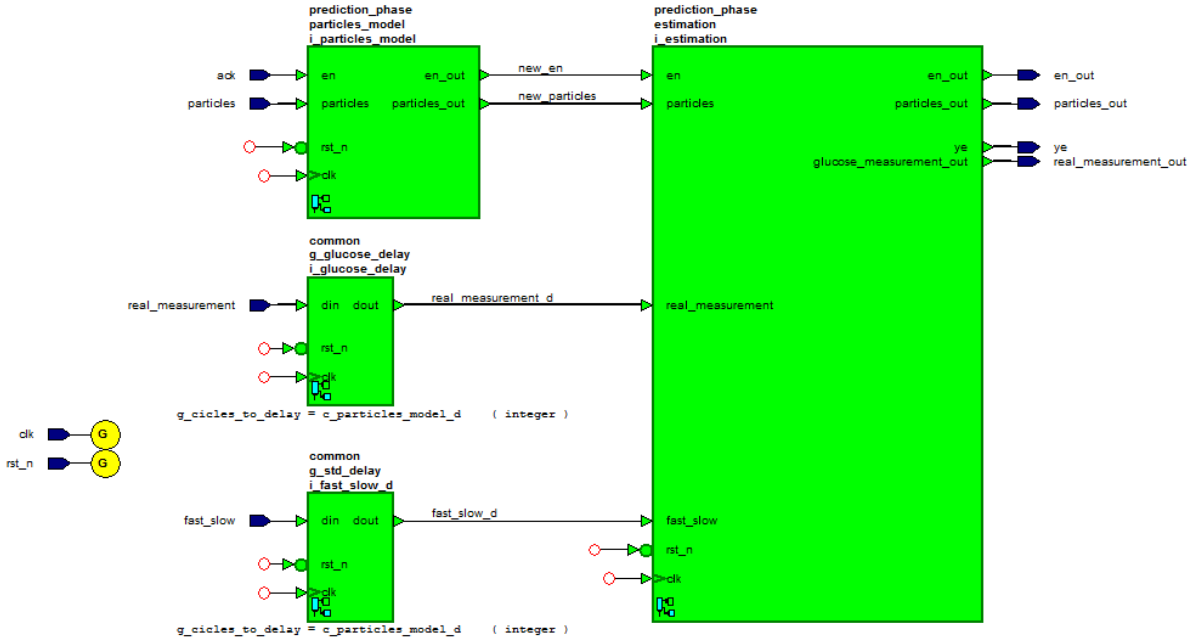


Figura 4.17 - Módulo prediction_phase.

4.7.1 Módulo particles_model

Este módulo se encarga de aplicar los modelos dinámicos explicados en la sección 1.3.1. a todas las partículas. En la Tabla 4.10 se muestran los puertos de este módulo.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activa a baja.
en	*std_logic*	entrada	Señal que indica que las partículas de entrada son válidas.
particles	*t_particles*	entrada	*Grid* de partículas sobre el que aplicar los modelos.
particles_out	*t_particles*	salida	Nuevo *grid* de partículas con los modelos aplicados.
en_out	*std_logic*	salida	Señal que indica que el nuevo *grid* de partículas es válido.

Tabla 4.10 - Interfaz módulo particles_model.

Tal y como se observa en la Figura 4.18 este módulo cuenta, por un lado, con un proceso (*for generate*) que genera tantas instancias como partículas del módulo *particle_model*. Esto se debe a que el módulo *particle_model* aplica los modelos dinámicos a una partícula concreta del *grid*. Es decir, los modelos se aplican en paralelo a todas las partículas.

Por otro lado, cuenta con un proceso que se encarga de registrar las salidas temporales (véase la Figura 4.19). Este proceso modela un registro que carga la entrada *particles_tmp* cuando su otra entrada *en_tmp* no es cero. Eso es así, puesto que las instancias del módulo *particle_model* trabajan de forma simultánea y todas terminan a la vez. Es decir, cuando una genere un uno en su señal *en_tmp*, el resto también lo hará.

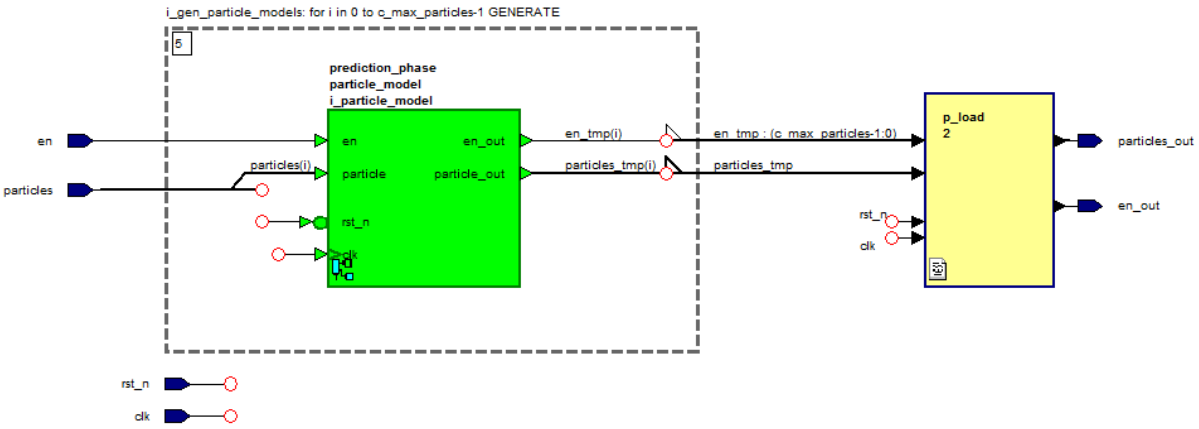


Figura 4.18 - Módulo particles_model.

```
-- p_load 2
p_load: process(clk, rst_n)
begin
  if rising_edge( clk ) then
  if rst_n = '0' then
    for i in 0 to c_max_particles-1 loop
      for j in 0 to c_particle_dimensions-1 loop
        particles_out(i)(j) <= ( others => '0' );
      end loop;
    end loop;
    en_out <= '0';
  else
      if en_tmp /= C_ZERO then
        particles_out <= particles_tmp;
        en_out <= '1';
      else
        en_out <= '0';
      end if;
    end if;
  end if;
end process;
```

Figura 4.19 - Proceso de guardado en particles_model

4.7.2 Módulo particle_model

El módulo *particle_model* (ver Figura 4.20) se encarga de aplicar los modelos dinámicos a una partícula dada. En la tabla que se muestra a continuación se observa la interfaz del módulo *particle_model*.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activa a baja.
en	*std_logic*	entrada	Señal que indica que las partículas de entrada son válidas.
particle	*t_particle*	entrada	Partícula sobre la que aplicar los modelos.
particle_out	*t_particle*	salida	Nuevas partículas con los modelos aplicados.
en_out	*std_logic*	salida	Señal que indica que la nueva partícula es válida.

Tabla 4.11 – Interfaz del módulo particle_model.

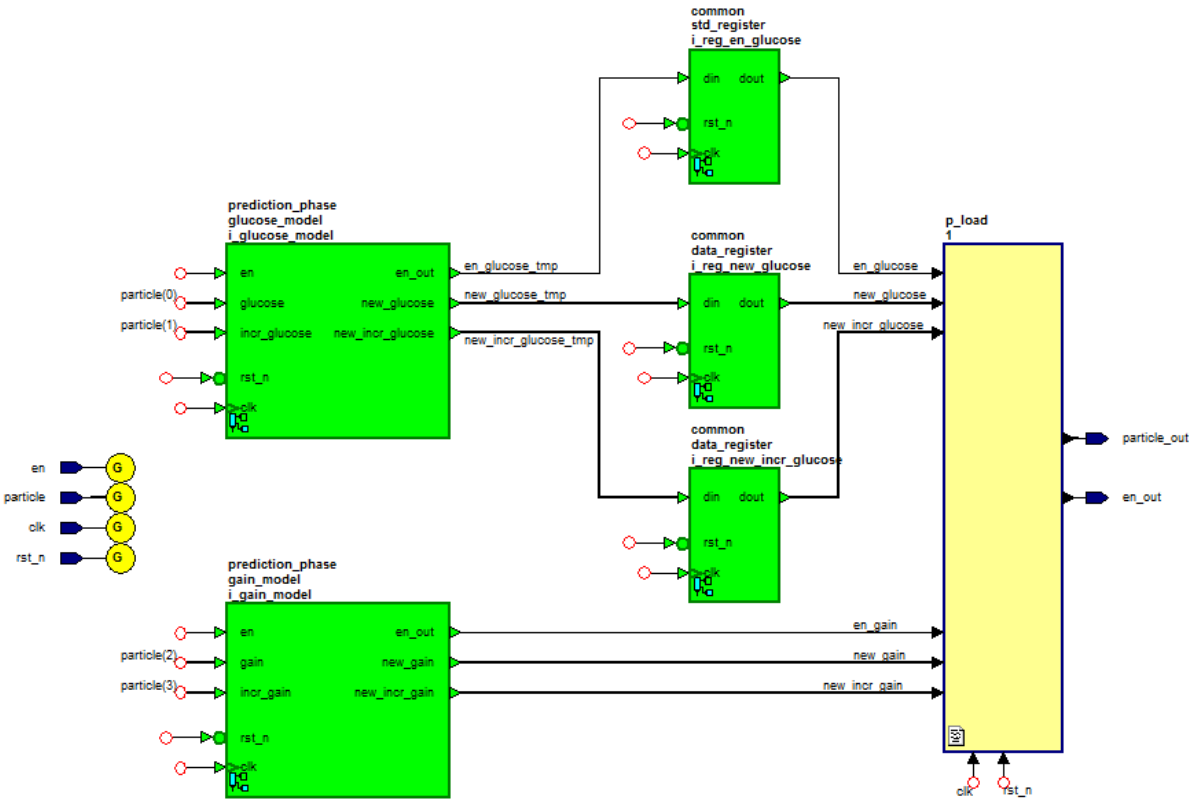


Figura 4.20 - Módulo particle_model.

Este módulo se encarga de aplicar los modelos de la glucosa y de la ganancia del sensor (mediante dos módulos que se explicarán más adelante) para posteriormente almacenarlos utilizando el proceso *p_load* (véase la Figura 4.21). Tal y como se observa, los módulos *glucose_model* y *gain_model*, aparte de aplicar los modelos, generan una salida que indica que los datos son válidos (*en_glucose* y *en_gain*). De esta forma, el proceso *p_load*, únicamente cargará los cálculos si ambas señales están a uno, es decir, si ambos módulos han realizado sus cálculos.

El módulo *glucose_model* (que se explica en detalle en la siguiente sección) tarda un ciclo de reloj menos que el módulo *gain_model*. Es por esto que las salidas del *glucose_model* se registran.

```
-- p_load 1
p_load: process(clk, rst_n)
begin
  if rising_edge( clk ) then
  if rst_n = '0' then
    particle_out(0) <= ( others => '0' );
    particle_out(1) <= ( others => '0' );
    particle_out(2) <= ( others => '0' );
    particle_out(3) <= ( others => '0' );
    en_out <= '0';
  else
      if en_glucose = '1' and en_gain = '1' then
        particle_out(0) <= new_glucose;
        particle_out(1) <= new_incr_glucose;
        particle_out(2) <= new_gain;
        particle_out(3) <= new_incr_gain;

        en_out <= '1';
      else
        en_out <= '0';
      end if;
    end if;
  end if;
end process;
```

Figura 4.21 - Proceso de guardado en particle_model.

4.7.3 Módulo *glucose_model*

Este módulo se encarga de aplicar el modelo dinámico de proceso a la glucosa. Los puertos de este módulo pueden observarse en la tabla que se muestra.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activo a baja.
en	*std_logic*	entrada	Señal que indica que las entradas son válidas.
glucose	*t_data*	entrada	Glucosa de entrada.
incr_glucose	*t_data*	entrada	Incremento de la glucosa de entrada.
new_glucose	*t_data*	salida	Glucosa calculada aplicando el modelo dinámico.
new_incr_glucose	*t_data*	salida	Incremento de la glucosa calculada apicando el modelo dinámico.
en_out	*std_logic*	salida	Señal que indica que las salidas son válidas.

Tabla 4.12 – Interfaz del módulo glucosa_model.

Tal y como se observa en la Figura 4.22, este módulo cuenta con tres procesos y seis sub-módulos. El modelo que aplica esta entidad viene representado mediante las siguientes ecuaciones (tal y como se vio en la sección 1.3.1):

$$g_{k+1} = g_k + \Delta g_k \qquad \textbf{(4.1)}$$

$$\Delta g_{k+1} = \Delta g_k + \omega_{g,k} \qquad \textbf{(4.2)}$$

De esta forma, lo primero que cabe destacar es la generación del ruido aleatorio $\omega_{g,k}$. Esta generación se consigue con el módulo *signed_pseudo_random* que ya se explicó en la sección 4.5. Debido a que este módulo utiliza un *reset* activo a alta, su señal se ve alimentada con la señal *rst_n* negada. Esto se consigue utilizando el proceso *p_not_rst*. Por otro lado, cabe destacar que el número aleatorio generado tendrá una anchura de c_rnd_noise_integer_lenght + c_rnd_noise_decimal_lenght. Por último, observar que el número aleatorio se multiplica por el parámetro Qg.

Posteriormente, se encuentran dos sumadores que aplican los modelos correspondientes. Y, por último, se registran los resultados. En este proceso de registro, también se registra la señal *en*. Esto se realiza con el fin de que esta señal viaje en sincronía con los datos calculados.

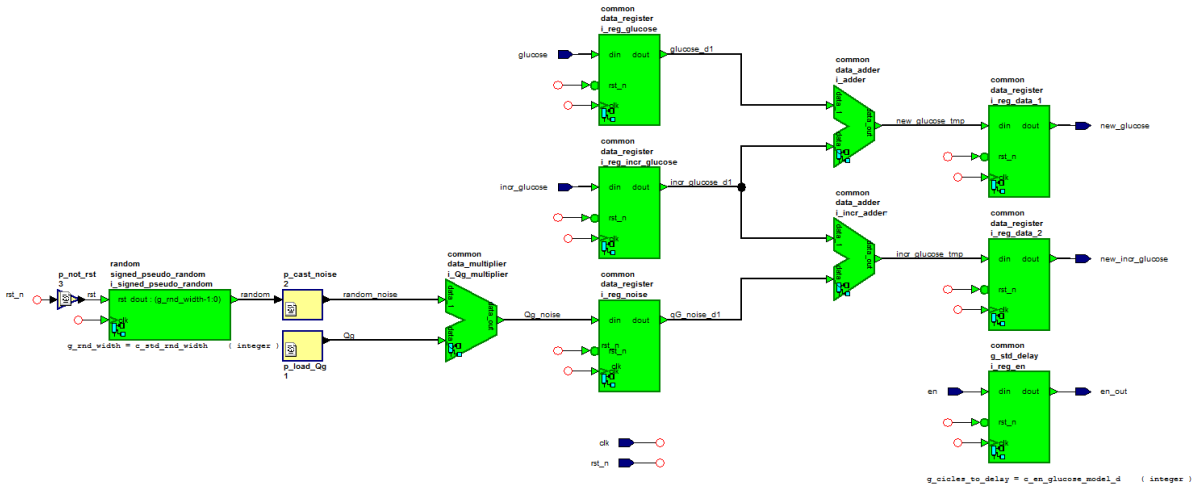


Figura 4.22 - Módulo glucose_model.

4.7.4 Módulo gain_model

Al igual que el módulo anterior (*glucose_model*), este módulo se encarga de aplicar los modelos dinámicos de proceso a la ganancia del detector. Sus interfaces se pueden observar en la Tabla 4.13.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activa a baja.
en	*std_logic*	entrada	Señal que indica que las entradas son válidas.
glucose	*t_data*	entrada	Glucosa de entrada.
incr_glucose	*t_data*	entrada	Incremento de la glucosa de entrada.
new_glucose	*t_data*	salida	Glucosa calculada aplicando el modelo dinámico.
new_incr_glucose	*t_data*	salida	Incremento de la glucosa calculada aplicando el modelo dinámico.
en_out	*std_logic*	salida	Señal que indica que las salidas son válidas.

Tabla 4.13 – Interfaz del módulo gain_model.

El diagrama de bloques de esta entidad puede observarse en la Figura 4.23. No es necesario explicar la funcionalidad de este módulo ya que es exactamente igual que el anterior.

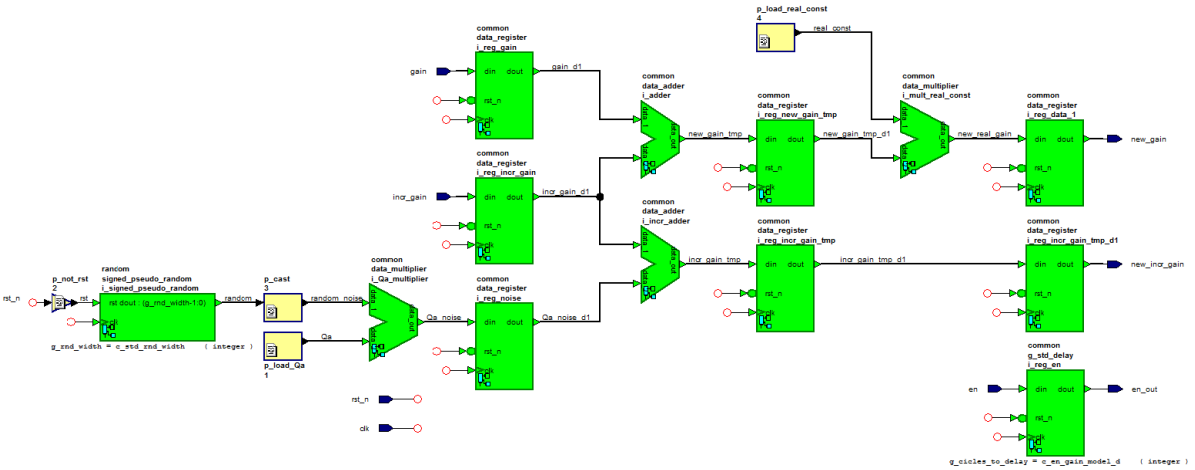


Figura 4.23 - Módulo gain_model.

4.7.5 Módulo *estimation*

Este módulo se encarga de aplicar los modelos de medición explicados en la sección 1.3.2. De esta forma se obtiene una medida estimada de la glucosa en sangre. La interfaz de este módulo se muestra en la Tabla 4.14.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activa a baja.
en	*std_logic*	entrada	Señal que indica que las entradas son válidas.
particles	*t_particles*	entrada	Partículas de entrada.
fast_slow	*std_logic*	entrada	Señal que indica el modelo a aplicar (rápido si vale 1, lento si vale 0).
real_measurement	*t_data*	entrada	Medida de glucosa real.
particles_out	*t_particles*	entrada	Partículas de salida (retrasadas N ciclos).
ye	*t_ye*	salida	Medida estimada de la glucosa en sangre aplicando el modelo correspondiente.
en_out	*std_logic*	salida	Señal que indica que las salidas son válidas.
glucose_measurement_out	*std_logic*	salida	Medida de la glucosa real de salida (retrasada N ciclos).

Tabla 4.14 - Interfaz módulo estimation.

En la Figura 4.24 se muestra el diagrama de bloques de este módulo. Esta entidad aplica los modelos de medición, generando así una medición estimada. Para aplicar los modelos de medición se utilizan los módulos *fast_model* y *slow_model* (que aplican el modelo rápido y el modelo lento respectivamente). Posteriormente se observa un multiplexor que se alimenta con la salida de estos dos módulos y que tiene como entrada de selección la señal *fast_slow_d*. Esta señal es el puerto *fast_slow* retrasado tantos ciclos como tarda el cálculo de las mediciones estimadas. De esta forma,

se escoge la medición estimada en función la señal *fast_slow*.

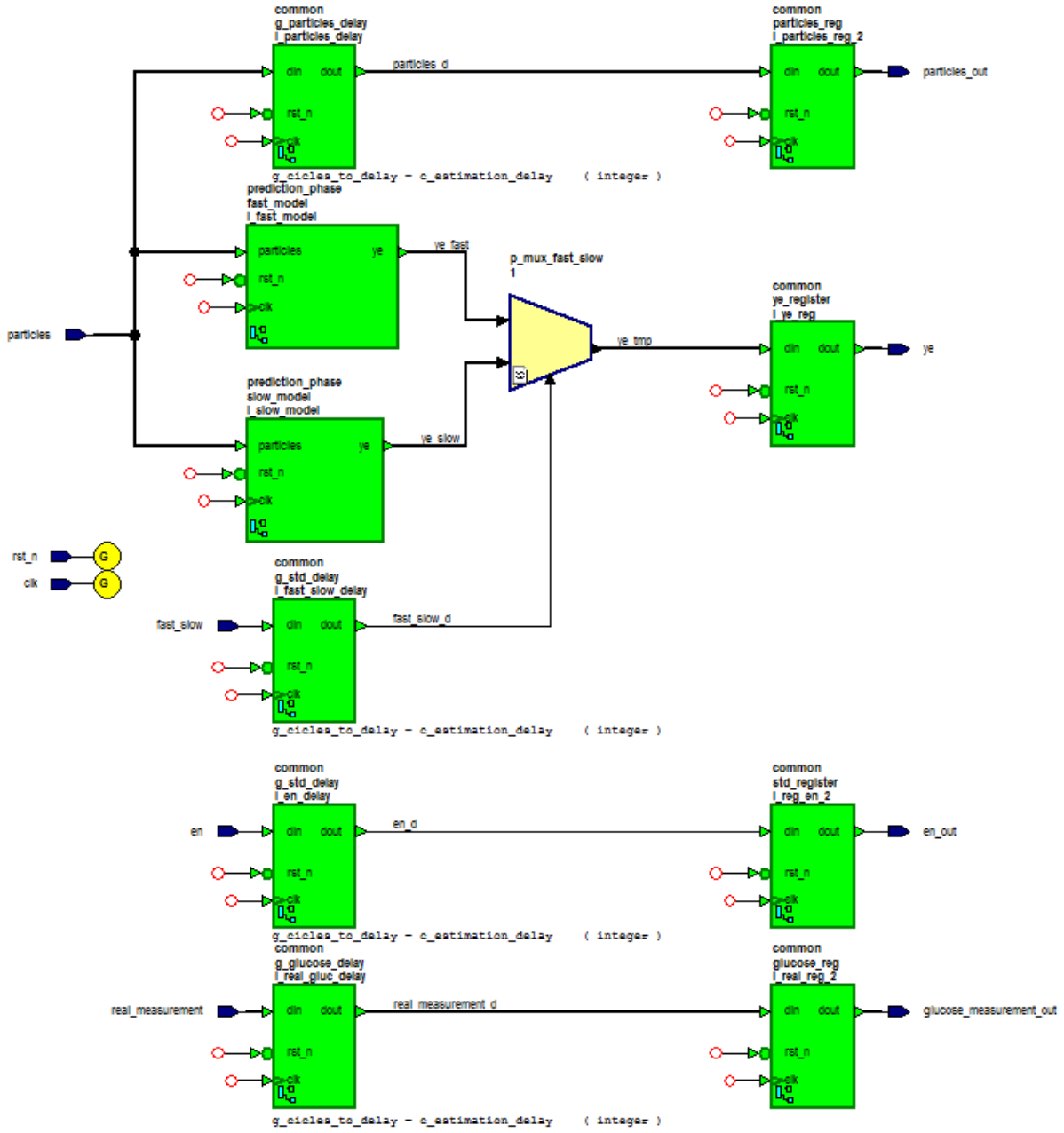


Figura 4.24 - Módulo estimation.

4.7.6 Módulo fast_model

Este módulo se encarga de aplicar el modelo rápido de la medición estimada a un grupo de partículas. Este modelo está representado por la siguiente ecuación (tal y como se vio en la sección 1.3.2):

$$ye_f = g * a + R_f * \omega_f \qquad \textbf{(4.3)}$$

Los puertos de este módulo se observan en la Tabla 4.15. Este módulo se implementa como un *for generate* que instancia a la entidad *particle_fast_model* tantas veces como partículas tiene el *grid* (véase la Figura 4.25). Esto se debe a que el módulo *particle_fast_model*, aplica el modelo rápido a una partícula dada.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activo a baja.
particles	*t_particles*	entrada	Partículas de entrada.
ye	*t_ye*	salida	Estimación de la glucosa en sangre aplicando el modelo rápido.

Tabla 4.15 - Interfaz módulo fast_model.

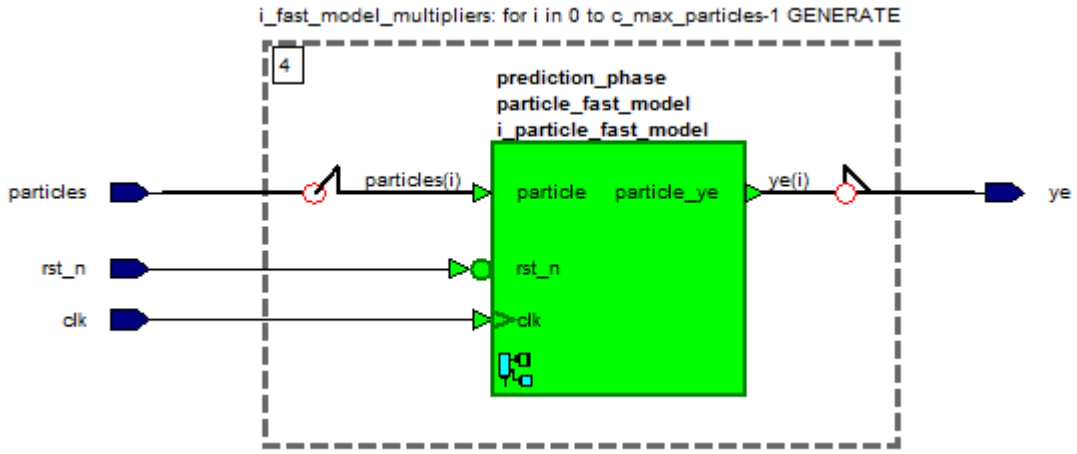


Figura 4.25 - Módulo fast_model.

El módulo *particle_fast_model* (representado en la Figura 4.26) se encarga de implementar el modelo de medición rápido a una partícula dada. Al igual que en el módulo *glucose_model*, se genera un número aleatorio utilizando el módulo *signed_pseudo_random* de tipo *t_data*. Este número aleatorio se multiplica por el parámetro Rf. Tal y como se explicó en la sección 1.3.2, este número aleatorio representa la covarianza a aplicar al ruido del modelo. Por otro lado, se multiplica la glucosa con la ganancia del sensor, obteniendo así la medición sin ruido. Tanto este resultado como el número aleatorio se registran y estos datos alimentan un sumador, cuya misión es aplicar el ruido al modelo. Por último, se registra este resultado, y se asigna al puerto *particle_reg*.

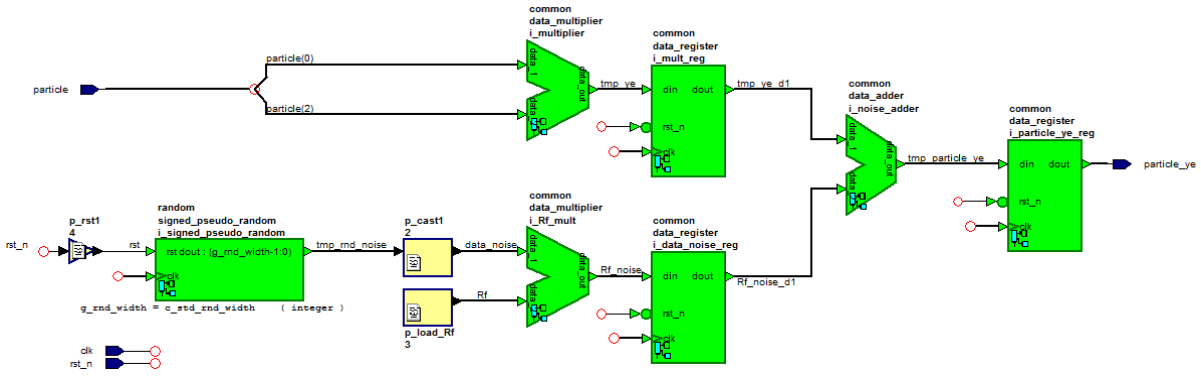


Figura 4.26 - Módulo particle_fast_model.

4.7.7 Módulo slow_model

Este módulo aplica el modelo lento de la medición estimada explicado en la sección 1.3.2, según la expresión:

$$ye_s = g + R_s * \omega_s$$

La interfaz de este módulo se muestra en la Tabla 4.16. Al igual que ocurre con el módulo *fast_model*, es *un for* generate que instancia al módulo *particle_slow_model* tantas veces como partículas tiene el *grid*. Este último módulo aparece representado en la Figura 4.27.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activo a baja.
particles	*t_particles*	entrada	Partículas de entrada.
ye	*t_ye*	salida	Estimación de la glucosa en sangre aplicando el modelo rápido.

Tabla 4.16 - Interfaz del módulo slow_model.

El módulo *particle_slow_model* se comporta de la misma manera que el módulo *particle_fast_model*. Su única diferencia es que donde el modelo rápido calculaba la multiplicación de la glucosa y de la ganancia del detector, en este modelo únicamente se registra la glucosa.

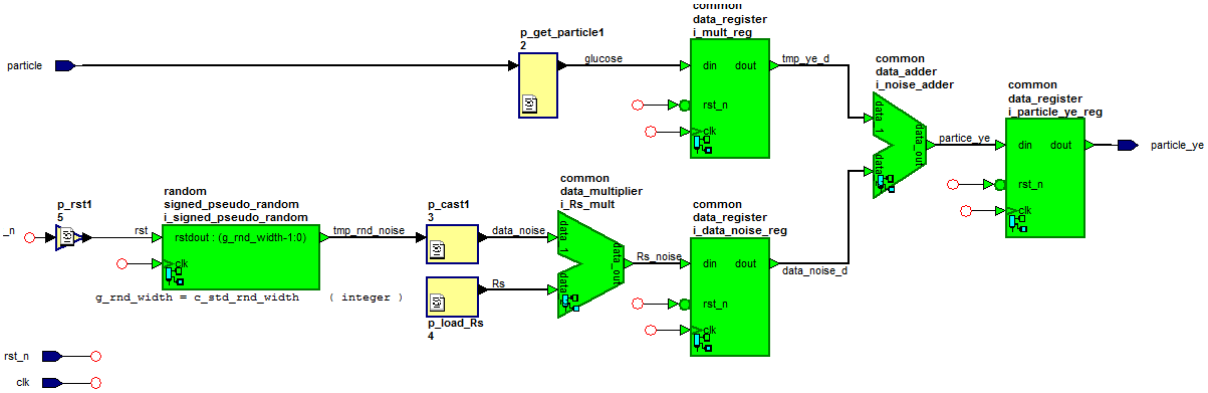


Figura 4.27 - Módulo particle_slow_model.

4.8 Librería *correction_phase*

En esta librería se encuentra el módulo *correction_phase* que se encarga de aplicar la fase de corrección del filtro de partículas. Los puertos del *top level* se muestran en la Tabla 4.17.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activa a baja.
en	*std_logic*	entrada	Esta señal indica cuándo los datos de entrada son válidos.
particles	*t_particles*	entrada	Partículas de entrada.
ye	*t_ye*	entrada	Estimación de la glucosa en sangre.
real_measurement	*t_data*	entrada	Medida real necesaria para el cálculo de los pesos de cada partícula.
en_out	*std_logic*	salida	Señal que indica cuándo las salidas son válidas.
weights	*t_weights*	salida	Array de pesos para cada partícula.
particles_out	*t_particles*	salida	Partículas de salida (retrasdas N ciclos).

Tabla 4.17 - Interfaz del módulo correction_phase.

El diagrama de bloques de este módulo se muestra en la Figura 4.28. En esta imagen se observa que este módulo se compone de tres sub-módulos. En primer lugar, se encuentra el módulo *PDF*, que se encarga de calcular la *PDF* y así obtener los pesos de las partículas. En segundo lugar, se encuentra el módulo *normalize_weights*, que normaliza los pesos.

En las siguientes sub-secciones se explica con más detalle cada uno de estos módulos.

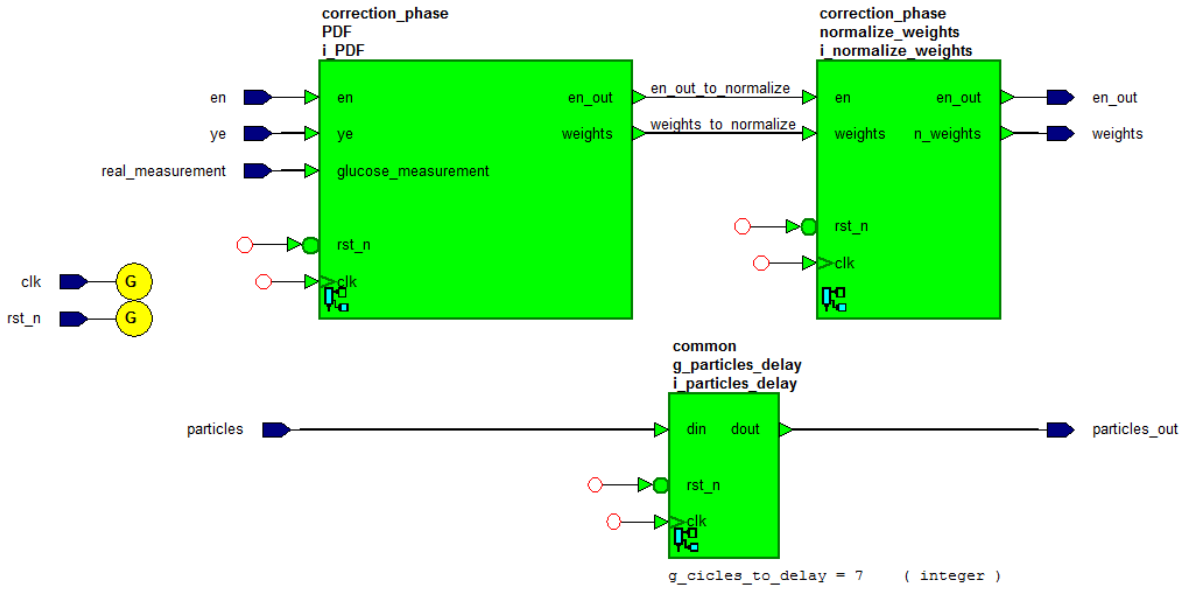


Figura 4.28 - Módulo correction_phase.

4.8.1 Módulo *PDF*

En la siguiente tabla se observa la interfaz del módulo *PDF*, que calcula la PDF a todas las partículas para obtener su peso.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activo a baja.
en	*std_logic*	entrada	Esta señal indica cuándo los datos de entrada son válidos.
ye	*t_ye*	entrada	Estimación de la glucosa en sangre.
glucose_measurement	*t_data*	entrada	Medida real necesaria para el cálculo de los pesos de cada partícula.
en_out	*std_logic*	salida	Señal que indica cuándo las salidas son válidas.
weights	*t_weights*	salida	Array de pesos para cada partícula.

Tabla 4.18 - Interfaz del módulo PDF.

Tal y como se observa en la

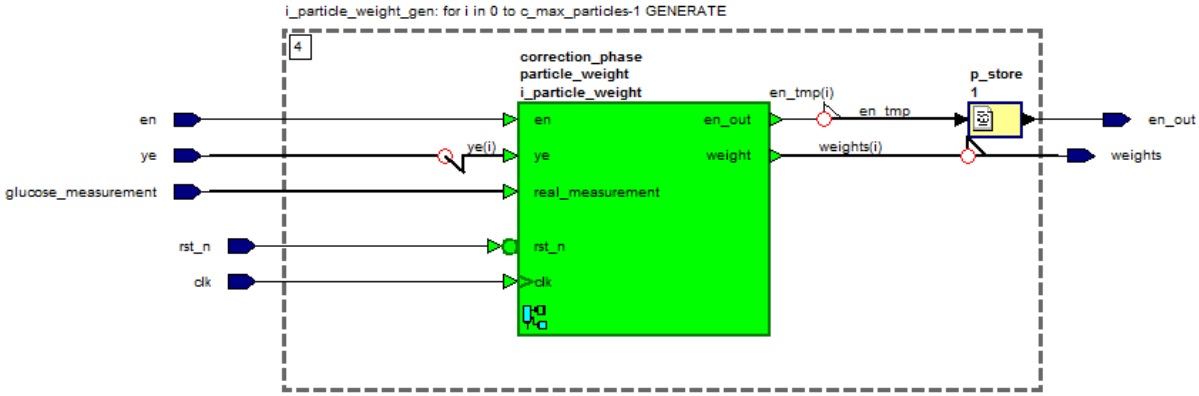


Figura 4.29, este módulo se compone de un proceso *for generate* que instancia al módulo *particle_weight* tantas veces como partículas tiene el *grid*. Tal y como se explicará en la siguiente sección, éste módulo se encarga de calcular el peso para una partícula dada. Se instancia un proceso que se encarga de generar la salida *en_out*. Este proceso es necesario ya que genera la salida *en_out* en base a las salidas *en_out_tmp* de cada uno de los módulos *particle_weight*.

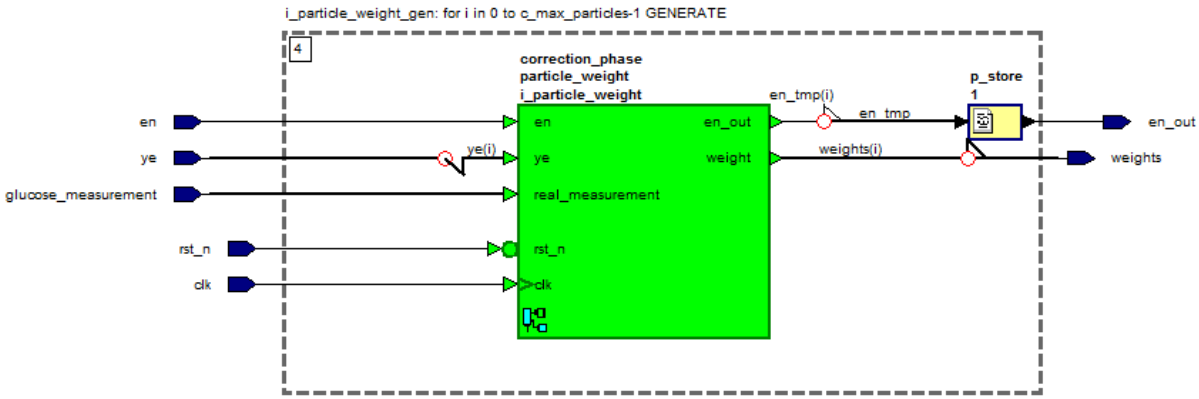


Figura 4.29 - Módulo PDF.

4.8.2 Módulo *particle_weight*

Este módulo es el encargado de calcular el peso para una partícula dada aplicando la PDF escogida. La interfaz de este módulo se muestra en la Tabla 4.19.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activo a baja.
en	*std_logic*	entrada	Esta señal indica cuándo los datos de entrada son válidos.
ye	*t_data*	entrada	Estimación de la glucosa en sangre para una partícula en concreto.
real_measurement	*t_data*	entrada	Medida real para una partícula en concreto.
en_out	*std_logic*	salida	Señal que indica cuándo las salidas son válidas.
weight	*t_weight*	salida	Peso calculado para la partícula.

Tabla 4.19 - Interfaz del módulo particle_weight.

En la Figura 4.30 se observa el diagrama de bloques de este módulo. La siguiente ecuación representa la PDF definida en la sección 3.3:

$$P = 1 \; si \; Y_{real} \leq Y_{predicha} < Y_{real} + \delta$$
$$P = 0 \; en \; caso \; contrario \qquad \textbf{(4.4)}$$

En primer lugar, el módulo *data_substractor* calcula la diferencia entre la medida real y la medida estimada. A continuación, se calcula el valor absoluto de esta diferencia con el módulo *data_abs*. Tanto este resultado, como el puerto de entrada *en*, se registran. Produciendo así la señal *abs_est_sub_r*.

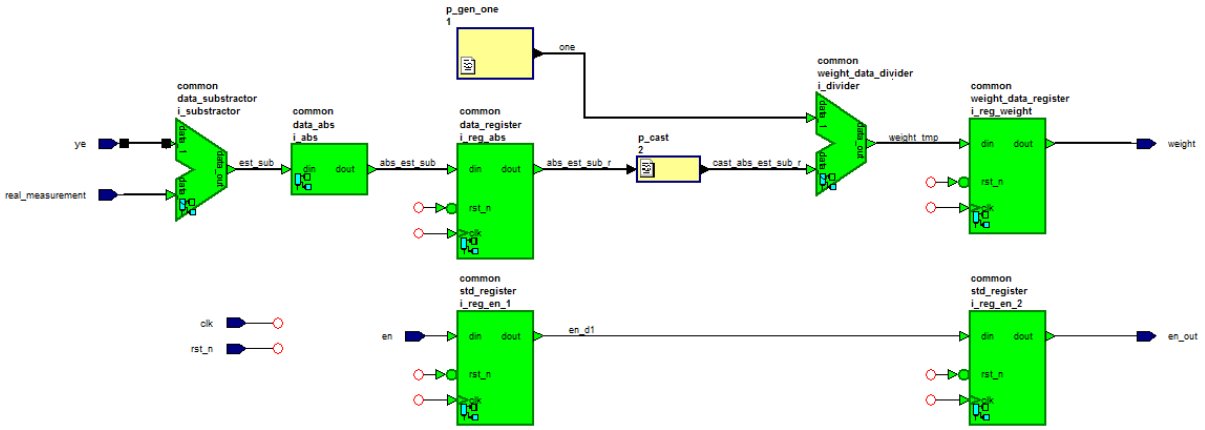


Figura 4.30 - Módulo particle_weight.

A la misma altura de este registro, se encuentra un proceso (*p_reg_one*) que genera un uno en aritmética de punto fijo con las mismas dimensiones que el tipo *t_weight_data* (véase la Figura 4.31). Esto se debe a que se debe calcular la inversa de la señal obtenida tras el valor absoluto. Por otro lado, se realiza una conversión al tipo *t_weight_data* a la señal *abs_est_sub_r*.

```
-- p_gen_one 1
one <= to_sfixed( 1.0, one'high, one'low );
```

Figura 4.31 - Generación de uno en aritmética de punto fijo.

De esta forma, se puede calcular la inversa de la diferencia utilizando el módulo *weight_data_divider*. Por último, se registra tanto el resultado (que será el peso resultante) como la señal *en_d1*, que generará el puerto *en_out*.

4.8.3 Módulo *normalize_weights*

Tras el cálculo de los pesos, estos se normalizan. Este módulo es el que se encarga de dicha normalización. La interfaz de módulo se muestra en la Tabla 4.20.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activo a baja.
en	*std_logic*	entrada	Esta señal indica cuándo los datos de entrada son válidos.
weights	*t_weights*	entrada	Array de pesos a normalizar.
en_out	*std_logic*	salida	Señal que indica cuándo las salidas son válidas.
n_weights	*t_weights*	salida	Pesos normalizados.

Tabla 4.20 - Interfaz del módulo normalize_weights.

El diagrama de bloques de este módulo se puede ver en la Figura 4.32. Tal y como se observa, por un lado, se realiza la suma de los pesos utilizando el módulo *weights_summation*, que se explica en la siguiente sección. Tras la suma de los pesos, se divide la misma por cada uno de los pesos. De esta forma se obtiene la normalización de cada uno de los pesos.

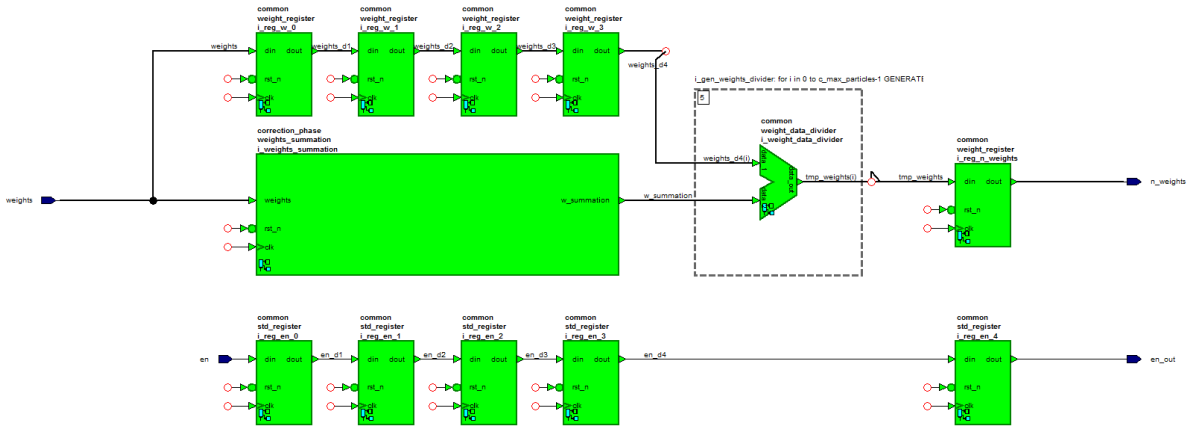


Figura 4.32 - Módulo normalize_weights.

4.8.4 Módulo *weights_summation*

Por último, el módulo *weights_summation*, realiza la suma de un *array* de diez posiciones utilizando una red de sumadores y registros tal y como se observa en la Figura 4.33.

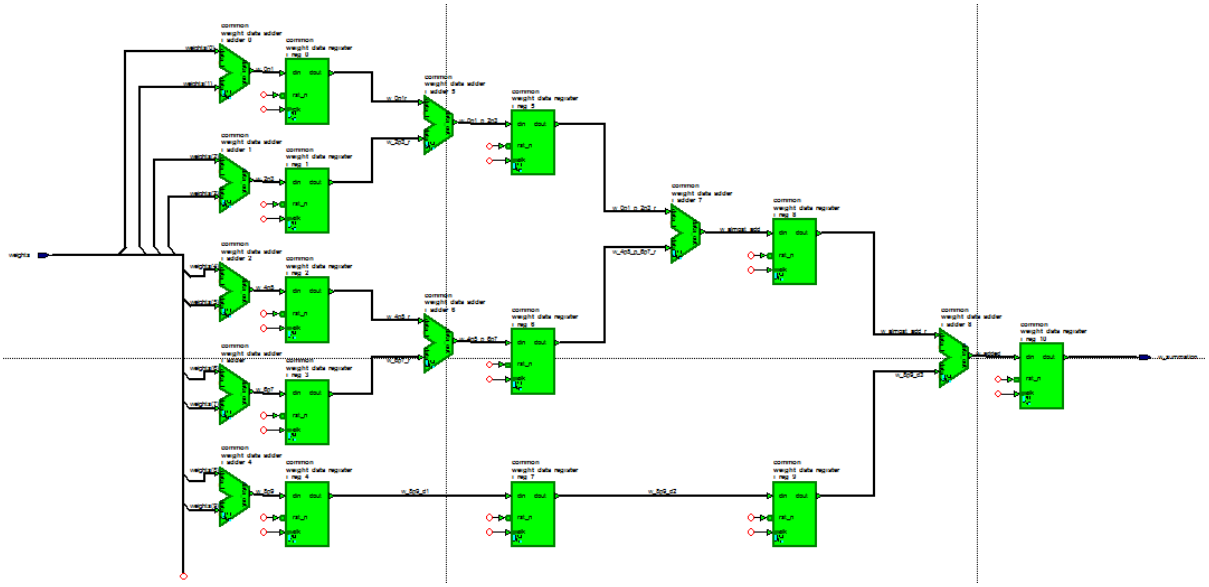


Figura 4.33 - Módulo weights_summation.

4.9 Librería *resampling_phase*

La última etapa del algoritmo del filtro de partículas es la etapa de *remuestreo*. El *top level* de esta librería es el módulo *resampling_phase*. En la Tabla 4.21 se observa la interfaz de este módulo.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activo a baja.
en	*std_logic*	entrada	Esta señal indica cuándo los datos de entrada son válidos.
particles	*t_particles*	entrada	*Grid* de partículas.
weights	*t_weights*	entrada	*Array* de pesos normalizados.
write	*std_logic*	salida	Señal que indica, por un lado, cuándo las nuevas partículas son válidas. Y, por otro lado, cuándo escribirlas en el módulo *particle_RAM*.
new_particles	*t_particles*	salida	Nuevo *grid* de partículas con aquellas que mejor peso han obtenido.

Tabla 4.21 - Interfaz del módulo resampling_phase.

Tal y como se observa en la Figura 4.34, este módulo únicamente instancia el algoritmo de torneo, que se explica en la siguiente sección.

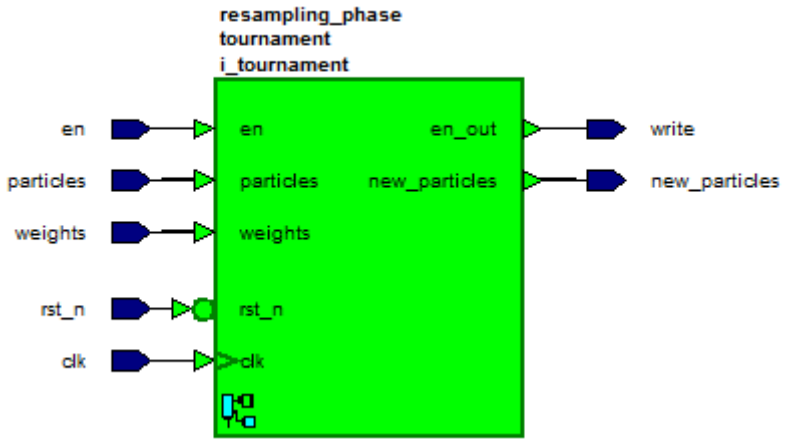


Figura 4.34 - Módulo resampling_phase.

4.9.1 Módulo *tournament*

El módulo *tournament* se encarga de escoger aquellas partículas con mejor peso y generar con ellas un nuevo *grid* de las mismas dimensiones que el conjunto de partículas de entrada. La interfaz de este módulo es idéntica a la del módulo *resampling_phase* (véase la Figura 4.21).

En la Figura 4.35 se observa que este módulo tiene tres entidades. Por un lado, instancia tantas veces como partículas existen el módulo *particle_tournament* (que se explica con más detalle en la siguiente sección). Esto se realiza de esta forma con el fin de que cada uno de estos módulos genere una de las partículas del *grid*.

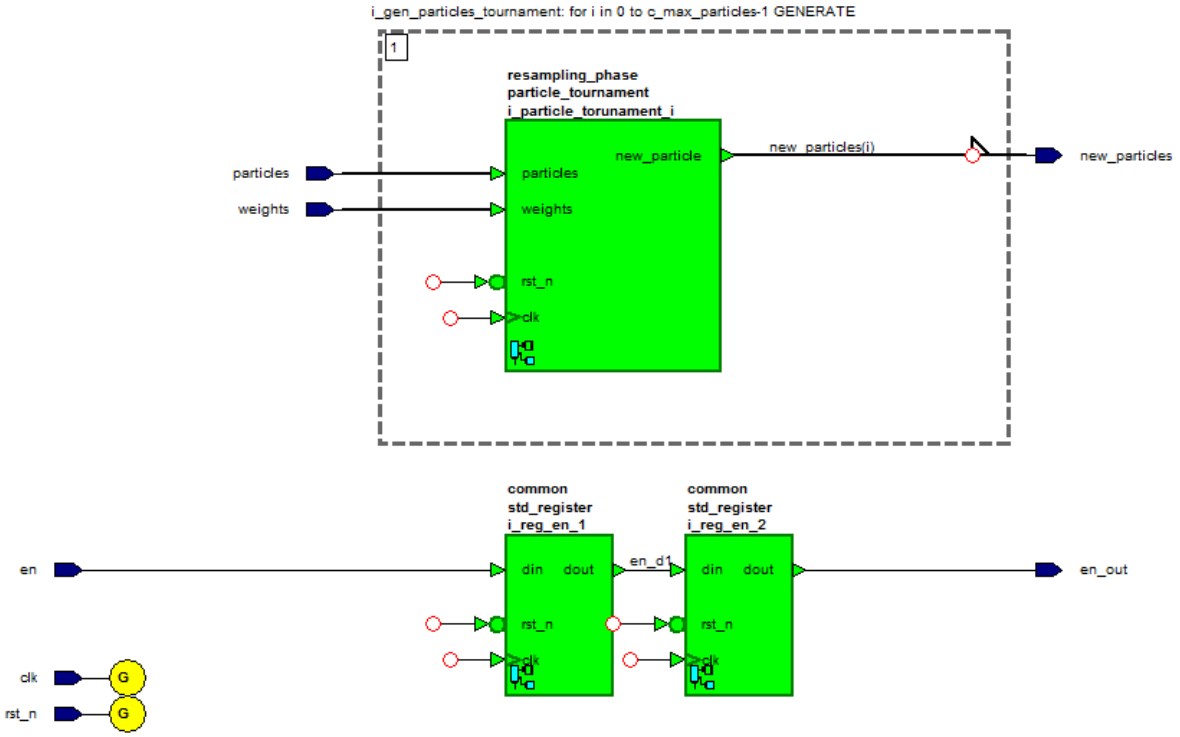


Figura 4.35 - Módulo tournament.

4.9.2 Módulo *particle_tournament*

Este módulo genera una partícula en base a los pesos y a las partículas que recibe como entrada (véase la Tabla 4.22).

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activo a baja.
en	*std_logic*	entrada	Esta señal indica cuándo los datos de entrada son válidos.
particles	*t_particles*	entrada	*Grid* de partículas.
weights	*t_weights*	entrada	*Array* de pesos normalizados.
write	*std_logic*	salida	Señal que indica, por un lado, cuándo las nuevas partículas son válidas. Y, por otro lado, cuándo escribirlas en el módulo *particle_RAM*.
new_particles	*t_particles*	salida	Nuevo *grid* de partículas con aquellas que mejor peso han obtenido.

Tabla 4.22 - Interfaz del módulo particle_tournament.

Tal y como se observa en la Figura 4.36, este módulo genera dos números aleatorios (*prnumber1* y *prnumber2*). A estos dos números se les realiza una traducción utilizando los procesos *p_translation_random_1* y *p_translation_random_2*, que se explican más adelante. Por otro lado, se registran las entradas que representan a las partículas y a los pesos. Esto es así ya que tanto los números aleatorios como estas dos entradas alimentan al proceso *p_particles_tournament*.

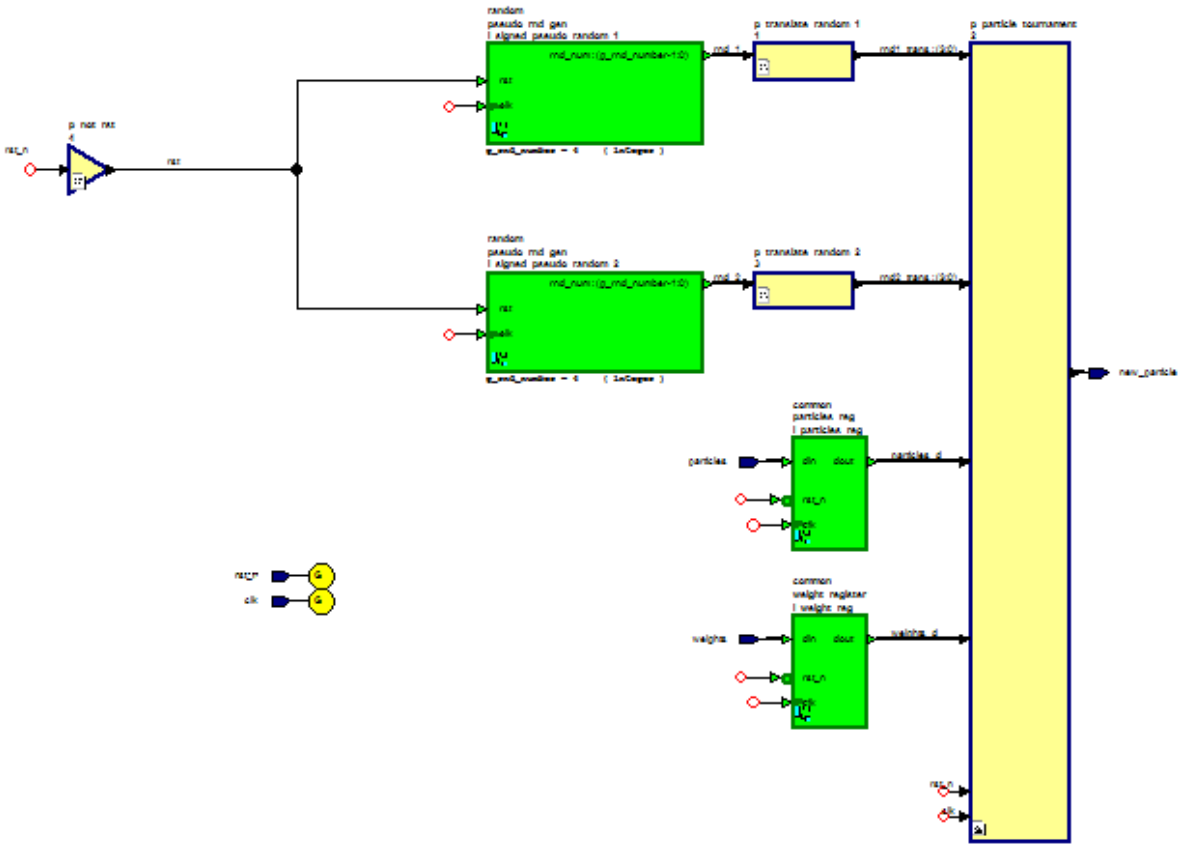


Figura 4.36 - Módulo particle_tournament.

En primer lugar, destacar que los procesos de traducción *p_translate_random_1* y *p_translate_random_2* se comportan de manera equivalente (véase la Figura 4.37). La entrada a este proceso es un número *pseudo*-aleatorio de una anchura de 4 *bits*, es decir, es un número decimal sin signo aleatorio que varía entre 15 y 0. Este número aleatorio indica una partícula del sistema, debido a que con la configuración actual se cuenta con 10 partículas, este número debe estar en el rango de 9 a 0. Es por eso que este proceso traduce los valores tal y como se muestra en la Figura 4.37.

```
-- p_translate_random_1 1
process(rnd_1)
begin
  case rnd_1 is
    when "0000" => rnd1_trans <=   "0000";
    when "0001" => rnd1_trans <=   "0001";
    when "0010" => rnd1_trans <=   "0010";
    when "0011" => rnd1_trans <=   "0011";
    when "0100" => rnd1_trans <=   "0100";
    when "0101" => rnd1_trans <=   "0101";
    when "0110" => rnd1_trans <=   "0110";
    when "0111" => rnd1_trans <=   "0111";
    when "1000" => rnd1_trans <=   "1000";
    when "1001" => rnd1_trans <=   "1001";
    when "1010" => rnd1_trans <=   "0000";
    when "1011" => rnd1_trans <=   "0001";
    when "1100" => rnd1_trans <=   "0010";
    when "1101" => rnd1_trans <=   "0011";
    when "1110" => rnd1_trans <=   "0100";
    when "1111" => rnd1_trans <=   "0101";
    when others => rnd1_trans <=   "0000";
  end case;
end process;
```

Figura 4.37 – Proceso p_translate_random.

El proceso *p_particles_tournament* (véase la Figura 4.38) se comporta como un registro en el que se guardan las nuevas partículas. En cada ciclo de reloj se comparan los pesos de dos partículas que son seleccionadas de forma aleatoria (mediante los números *rnd_1* y *rnd_2*) utilizando el generador de números aleatorios de la sección 4.5. En caso de que el primer peso sea mayor o igual al segundo, se escoge la partícula de la posición indicada por el primer número aleatorio. En caso contrario, se escoge la partícula indicada por el segundo número aleatorio.

```
-- p_particle_tournament 1
p_particle_tournament: process( clk, rst_n )
begin
  if ( rising_edge( clk ) ) then
    if rst_n = '0' then
    for i in 0 to c_particle_dimensions-1 loop
      new_particle(i) <= (others => '0' );
    end loop;
  else
      if ( weights_d( to_integer( signed ( rnd_1 ) ) ) >= weights_d( to_integer( signed ( rnd_2 ) ) ) ) then
          new_particle <= particles_d( to_integer( signed ( rnd_1 ) ) );
      else
          new_particle <= particles_d( to_integer( signed ( rnd_2 ) ) );
      end if;
    end if;
  end if;
end process;
```

Figura 4.38 - Proceso p_particles_tournament.

4.10 Librería *particle_RAM*

Por último, en el módulo *particle_RAM* se almacena la población de partículas con la que se trabaja durante una iteración. Este módulo alimenta a la etapa de predicción, y a su vez, es alimentado por la fase de *remuestreo*. Tal y como se observa en la Tabla 4.23, este módulo cuenta con dos puertos de

entrada cuya finalidad en inyectar datos desde un *testbench* externo al sistema. Cabe destacar que las entradas del *testbench* se consideran prioritarias a las calculadas por la etapa de *remuestreo*.

Nombre	Tipo	Sentido	Comentario
clk	*std_logic*	entrada	Reloj del sistema.
rst_n	*std_logic*	entrada	Señal de *reset* activo a baja.
write	*std_logic*	entrada	Señal que indica que se tiene un nuevo *grid* de partículas procedente de la etapa de *remuestreo*.
particles_to_wr	*t_particles*	entrada	*Grid* de partículas procedente de la etapa de *remuestreo*.
req_from_tb	*std_logic*	entrada	Señal que indica que se tiene un nuevo *grid* de partículas procedente de la etapa de *testbench*.
particles_from_tb	*t_particles*	entrada	*Grid* de partículas procedente de la etapa de *testbench*.
particles_to_read	*t_particles*	salida	Nuevo *grid* de partículas a usar en los cálculos. Esta salida se le proporciona a la etapa de predicción.
req	*std_logic*	salida	Señal que indica que hay un nuevo *grid* de partíclas para la etapa de predicción.

Tabla 4.23 - Interfaz del módulo particle_RAM.

Por otro lado, en la Figura 4.39 se observa que la generación de la señal de salida *req* consiste en registrar el resultado de una *or* lógica de las señales de escritura (tanto la que viene desde la etapa de *remuestreo* como la que viene desde el *testbench*).

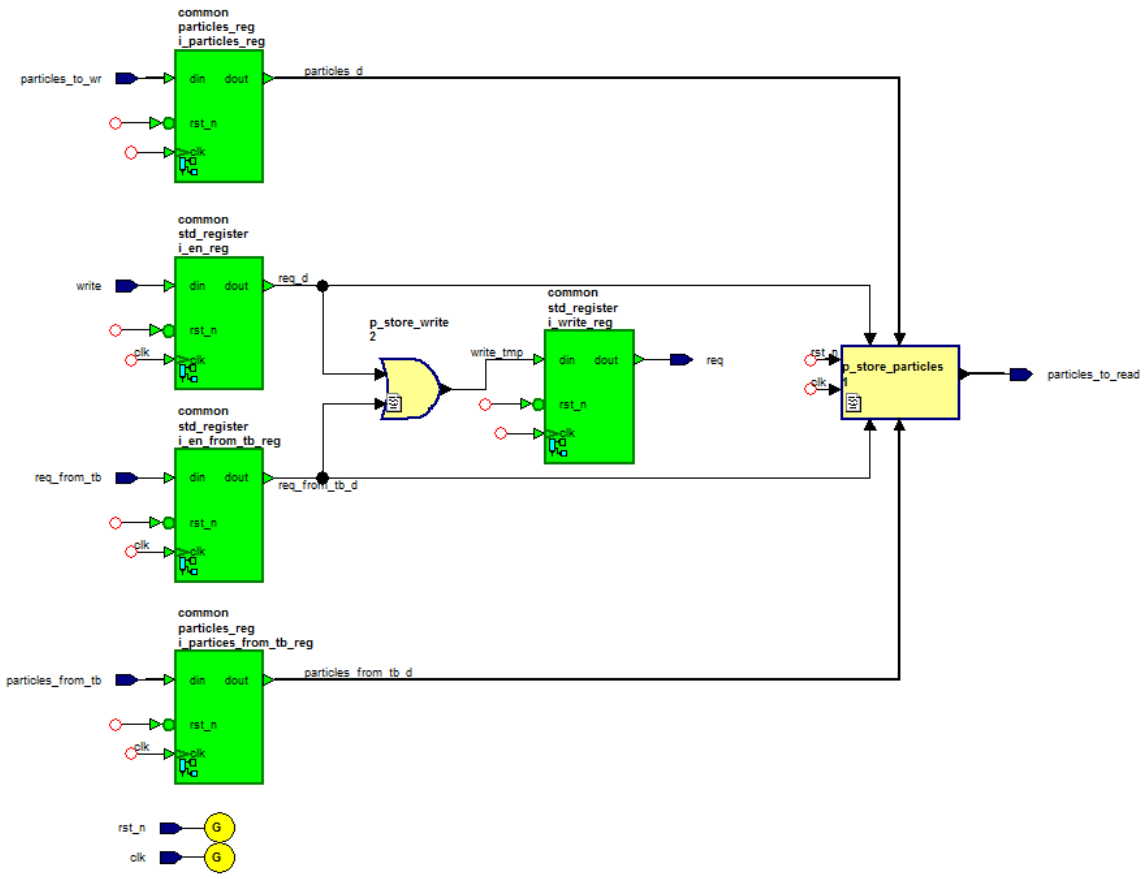


Figura 4.39 - Módulo particle_RAM.

En cuanto a la generación del *grid* de partículas, se utiliza el proceso *p_store_particles* (véase la Figura 4.40). En este proceso es donde se registra o bien el *grid* de partículas obtenido desde el *testbench*, o bien, el obtenido desde la etapa de *remuestreo*.

```
-- p_store_particles 1
p_store_particles: process(clk, rst_n)
begin
  if rising_edge( clk ) then
    if rst_n = '0' then
      for i in 0 to c_max_particles-1 loop
        for j in 0 to c_particle_dimensions-1 loop
          particles_to_read(i)(j) <= ( others => '0' );
        end loop;
      end loop;
    else
      if req_from_tb_d = '1' then
        particles_to_read <= particles_from_tb_d;
      elsif req_d = '1' then
        particles_to_read <= particles_d;
      end if;
    end if;
  end if;
end process;
```

Figura 4.40 - Proceso p_store_particles.

Capítulo 5

5. Síntesis

5.1 Configuración de la herramienta

Para realizar la síntesis del diseño en *HDL Designer* se ha utilizado la herramienta *XST* de *Xilinx ISE 14.1*. La configuración utilizada para sintetizar el diseño en *HDL Designer* es:

- *FPGA Vendor: Xilinx.*
- *Family: virtex6*
- *Device: xc6vlx2t40*
- *Package: 1ff1156*

5.2 Análisis del informe de síntesis

En la Tabla 5.1 se muestra la información resultante de realizar la síntesis, obtenida del *HDL Synthesis Report*.

En este mismo informe se detalla el resumen de la temporización (*Timing summary*) y el uso total de la memoria del diseño:

- Período mínimo: 135.632ns (Frecuencia máxima: 7.373 MHz).
- Tiempo mínimo de llegada de entradas antes del reloj: 1.840ns.
- Tiempo máximo necesario para las salidas después del reloj: 0.783ns.

Módulo	Número de componentes
Sumadores/Restadores	3089
Multiplicadores	60
Registros	2834
Comparadores	1300
Multiplexores	77947
Máquinas de estados	310
Xors	1270

Tabla 5.1 - Módulos utilizados en la síntesis del proyecto con diez partículas.

En la Figura 5.1 se muestra un resumen de la ocupación de la placa. En esta figura se observa que el diseño requiere una cantidad de *LUTs* (de sus siglas en ingés *Lookup tables*) mayor de la que proporciona la placa.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	50291	301440	16%
Number of Slice LUTs	152208	150720	100%
Number of fully used LUT-FF pairs	24136	178363	13%
Number of bonded IOBs	32	600	5%
Number of BUFG/BUFGCTRLs	3	32	9%
Number of DSP48E1s	120	768	15%

Figura 5.1 - Resumen de la utilización del dispositivo tras la síntesis.

Debido a que la placa no soporta el diseño actual, se realizaron unos simples ajustes con el fin de disminuir el número de *LUTs*, de esta forma, se sintetizó un diseño reducido. El cambio consistió en disminuir en uno el número de bits de los números aleatorios generados en la PDF. De esta forma, se obtuvieron los siguientes resultados de temporización:

- Período mínimo: 135.626ns (Frecuencia máxima: 7.373 MHz).
- Tiempo mínimo de llegada de entradas antes del reloj: 1.831ns.
- Tiempo máximo necesario para las salidas después del reloj: 0.783ns.

De la misma manera que para la síntesis anterior, en la Figura 5.2 se muestra un resumen de la ocupación de la placa. En este caso, se observa que la placa cuenta con suficiente espacio para implementar el diseño del filtro de partículas.

Device Utilization Summary (estimated values)			[-]
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	44095	301440	14%
Number of Slice LUTs	135021	150720	89%
Number of fully used LUT-FF pairs	21326	157790	13%
Number of bonded IOBs	32	600	5%
Number of BUFG/BUFGCTRLs	3	32	9%
Number of DSP48E1s	112	768	14%

Figura 5.2 - Resumen de la utilización del dispositivo tras la síntesis reducida.

5.3 Camino crítico

El informe de síntesis establece la siguiente ruta como camino crítico:

```
Data Path:
i_correction_phase/i_normalize_weights/i_weights_summation/i_reg_10/dout_-
20 to i_correction_phase/i_normalize_weights/i_reg_n_weights/dout_6_8

    ----------------------------------------
    Total                     135.626ns (54.166ns logic, 81.460ns route)
                                        (39.9% logic, 60.1% route)
```

El inicio de este es la salida del módulo *weights_summation*, que acumula los valores de los pesos de todas las partículas. Esta salida entra como segundo operando al divisor *i_weight_data_divider*, y al salir de éste se inyecta como entrada al registro *i_reg_n_weights*. En la Figura 5.3 se ilustran los módulos que intervienen en el camino crítico.

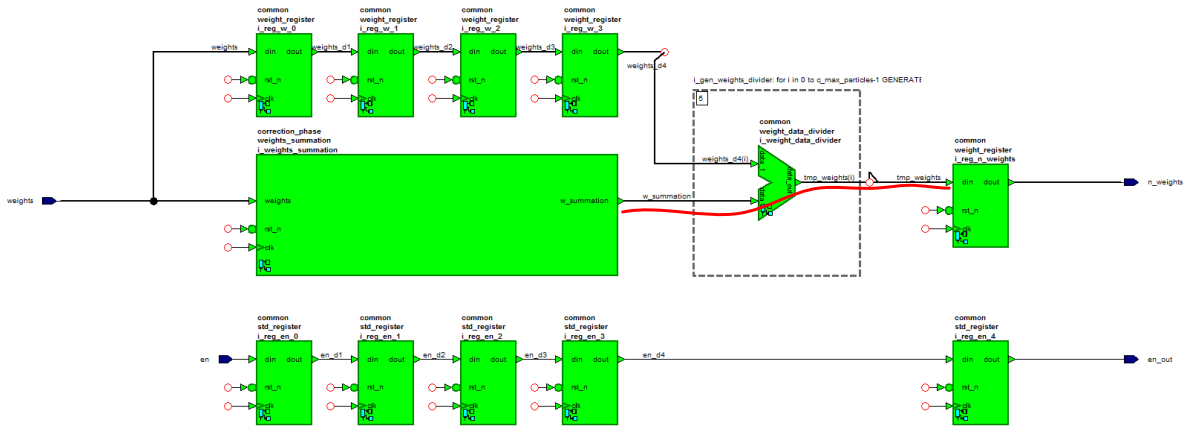


Figura 5.3 - Camino crítico del diseño hardware del filtro de partículas.

Capítulo 6

6 Resultados experimentales

En este capítulo se detallan los resultados obtenidos mediante las simulaciones del sistema utilizando unas mediciones sintéticas. Para crear las representaciones gráficas se ha seguido el siguiente procedimiento:

En primer lugar, se ha desarrollado un *testbench* utilizando la herramienta *HDL Designer*. En él se encuentran dos arquitecturas:

- El componente del filtro de partículas.
- Un componente capaz de escribir en un fichero los resultados obtenidos por el filtro.

En segundo lugar, utilizando la herramienta *Matlab* se representan en figuras los resultados obtenidos por el filtro de partículas.

Y, por último, se ejecutan los *scripts Matlab* del filtro de partículas con la misma entrada sintética para comparar ambos resultados.

Tal y como se comentó en el capítulo 2, el vector de estado del sistema se compone de cuatro componentes. Los componentes a estudiar son el primero (la estimación de la glucosa en sangre) y el tercero (la estimación de la ganancia del sensor).

6.1 Estructura del *testbench*

En esta sección se explica la estructura del *testbench* que recoge los resultados del filtro de partículas. Tal y como se observa en la Figura 6.1, este *testbench* cuenta con una serie de bloques embebidos y dos componentes.

En primer lugar, los bloques embebidos *p_rst* y *p_clk* generan, respectivamente, la señal de *reset* y la señal del reloj. En segundo lugar, el bloque *p_real_stimuli* genera la entrada sintética de la medición de la glucosa en sangre. En este caso se genera una señal con un valor 97,34 constante. En tercer lugar, el proceso *p_fast_stimuli* genera la señal que indica al filtro si la medición que se le inyecta proviene de una medida lenta (pinchazo), o bien, de una medida rápida (obtenida a través de un MCG), tal y como se explicó en el capítulo 1. Por otro lado, el bloque *p_tb_stimuli* se encarga de generar la primera población de partículas, es decir, la etapa de inicialización del filtro de partículas (explicado en el capítulo 2).

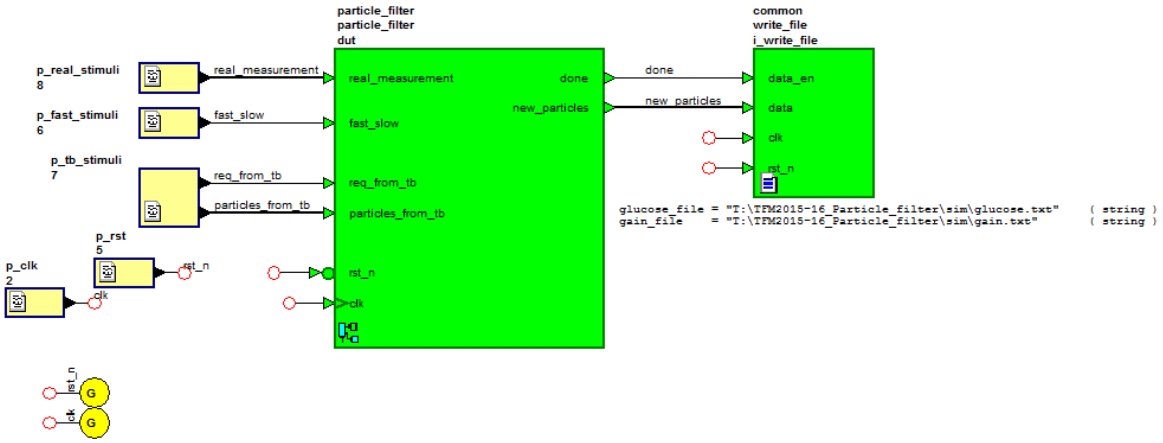


Figura 6.1 - Estructura del testbech tb_particle_filter.

En cuanto a los componentes instanciados, por un lado, se tiene el bloque del filtro de partículas (*particle_filter*), mientras que, por otro lado, se tiene el componente que escribe los ficheros de resultados (*write_file*). Este último componente, genera dos ficheros, uno con la estimación de la glucosa y otro con la estimación de la ganancia del sensor. Cabe destacar que, por cada estimación, este módulo escribe tantos resultados como partículas se tengan definidas.

6.2 Representación de resultados

Con el fin de representar los resultados obtenidos por este *testbench*, se desarrolló un *script Matlab* para leer los ficheros generados y mostrarlos por pantalla.

En la Figura 6.2 se observa la parte del *script* que lee el fichero generado por el *testbench* con la estimación de la glucosa. Este *script* lee línea a línea el archivo generado. De esta forma, acumula los valores leídos en la variable *particles*. Cada que vez ha consumido tantas líneas como partículas, guarda la media de la suma de estos valores (variable *particles*) en el *vector glucose* y limpia la variable *particles* asignándole el valor cero. De esta forma, finalizada la lectura se obtiene el vector *glucose*, que, para cada posición de este vector, almacena la media de los valores de la glucosa estimada para cada partícula. Destacar que los datos leídos del fichero se encuentran en binario, por eso es necesario convertir estos datos a la aritmética de punto fijo utilizando la función *bin2num* tal y como se muestra en la Figura 6.2.

Por otro lado, guarda los valores de cada partícula con el fin de representar el *grid* de partículas en cada muestreo. Esto se realiza con los vectores *y_particle_glucose* que almacena el valor de la glucosa de cada partícula, y *x_particle_glucose* que almacena el número de la partícula. De esta forma, se obtienen puntos cuya coordenada Y se corresponde con el valor de la glucosa de la partícula, y la coordenada X con el número de la partícula.

```
%-------------------------------------------------------------------------------
% Leemos el archivo de salidas de la glucosa y generamos la figura
%-------------------------------------------------------------------------------
fid = fopen('glucose.txt');
idx = 1;
particle_idx = 1;
number = 1;

particles_sum = fi(0, KF_t_ext, KFArith_ext);

quant.mode      = 'fixed';
quant.roundmode = 'ceiling';
quant.format    = [25 15];
KFFormat = quantizer(quant);

tline = fgetl(fid);
while ischar(tline)
  if(particle_idx <= 11)
    particles(particle_idx) = bin2num(KFFormat, tline);

    y_particle_glucose(number) = fi(particles(particle_idx), KF_t_ext, KFArith_ext);
    x_particle_glucose(number) = idx;
    number = number + 1;

    particles_sum = particles_sum + fi(particles(particle_idx), KF_t_ext, KFArith_ext);
    particle_idx = particle_idx + 1;
  end
  if ( particle_idx == 11 )
      glucose(idx) = particles_sum / 10;
      idx          = idx + 1;
      particles_sum = fi(0, KF_t_ext, KFArith_ext);
      particle_idx = 1;
  end
  tline        = fgetl(fid);
end

fclose(fid)
```

Figura 6.2 - Lectura del fichero glucose.txt.

Tras la lectura del fichero, primero se muestra el resultado por pantalla a través de una figura (tanto para el vector con los valores medios de la glucosa, como para los valores del *grid* de partículas), para a continuación guardar dichas figuras cada una en un archivo (véase la Figura 6.3).

```
%--------------------------------------------------------------------------
% Dibujamos la grafica
%--------------------------------------------------------------------------
figName = 'Estimated glucose';
fig1=figure(1);
hplot1 = plot(glucose);
xlabel('Sample');
ylabel('Glucose');
grid on;
saveas(fig1, figName, 'jpg');
close(fig1)


figName = 'Particles Glucose';
fig2=figure(2);
hplot1 = scatter(x_particle_glucose,y_particle_glucose);
xlabel('Sample');
ylabel('Particles Glucose');
grid on;
saveas(fig2, figName, 'jpg');
close(fig2)
```

Figura 6.3 - Representación de los resultados de la glucosa del testbench.

Por último, se realizan las mismas operaciones para el fichero con los resultados de la ganancia del sensor. De esta forma, se obtienen cuatro figuras con los resultados de la ejecución del *testbench*:

- Los valores medios de la glucosa estimada.
- Los valores de la glucosa para el *grid* de partículas.
- Los valores medios de la ganancia estimada.
- Los valores de la ganancia para el *grid* de partículas.

6.3 Resultados obtenidos

Tras lanzar el *testbench* y obtener las figuras de los resultados, se comparan con los resultados esperados en *Matlab*.

Para conseguir los resultados esperados, se ejecuta el *script Matlab run_particle_filter* (explicado en el capítulo 3) con una entrada sintética constante con valor 97,34 (el mismo valor con el que se lanzó el *testbench*, véase la sección 6.1).

En la Figura 6.4 se observa la estimación obtenida de la glucosa tras la ejecución del *script Matlab*. Puede observarse que se estima un valor constante en torno al valor 97 (valor de la entrada sintética).

Por otro lado, la Figura 6.5 muestra el resultado obtenido de la glucosa tras la ejecución del *testbench*. En este caso, se observa que el valor estimado de la glucosa varía principalmente entre valores de 95 y 100, que son muy cercanos al valor esperado (97,34). Sin embargo, se observan ciertos picos en la estimación de la glucosa.

Mientras que, la ganancia del sensor, representada por la Figura 6.6 para la calculada por el *script Matlab* y por la Figura 6.7 para la calculada a través del *testbench*, coinciden exactamente. Esto se

debe a que en ambos resultados se calcula el modelo con exactitud (incluyendo una constante de degradación tal y como se explicó en los capítulos 3 y 4).

Por último, en la Figura 6.8 se observan los valores de la glucosa para todas las partículas del *grid*. En esta imagen se representan 10 puntos para cada muestreo, es decir, el valor de la glucosa para las 10 partículas en cada muestreo. De esta forma se puede observar que para cada muestreo cada partícula toma un valor diferente debido al ruido aleatorio que se le aplica. Mientras que, en la Figura 6.9 se muestran los valores de la ganancia de la misma manera que la anterior imagen. Puesto que a estos valores no se les aplica ningún ruido, todas las partículas toman exactamente el mismo valor.

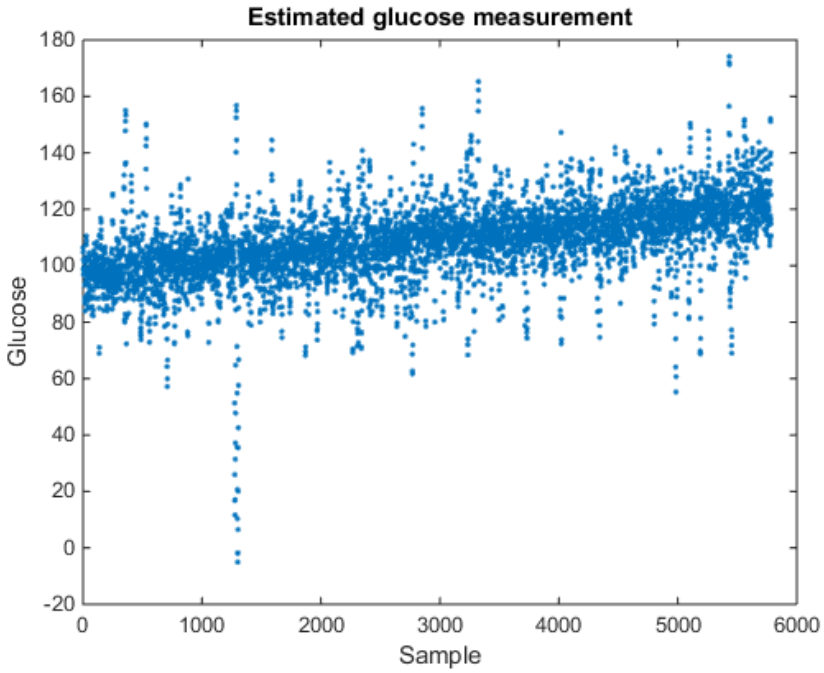


Figura 6.4 - Estimación Matlab de la glucosa.

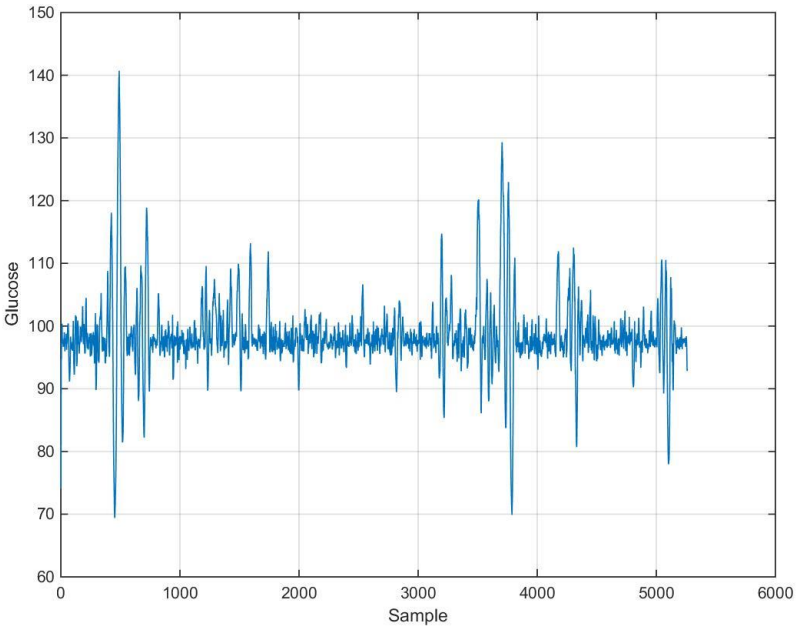


Figura 6.5 - Resultado de la glucosa con el testbench.

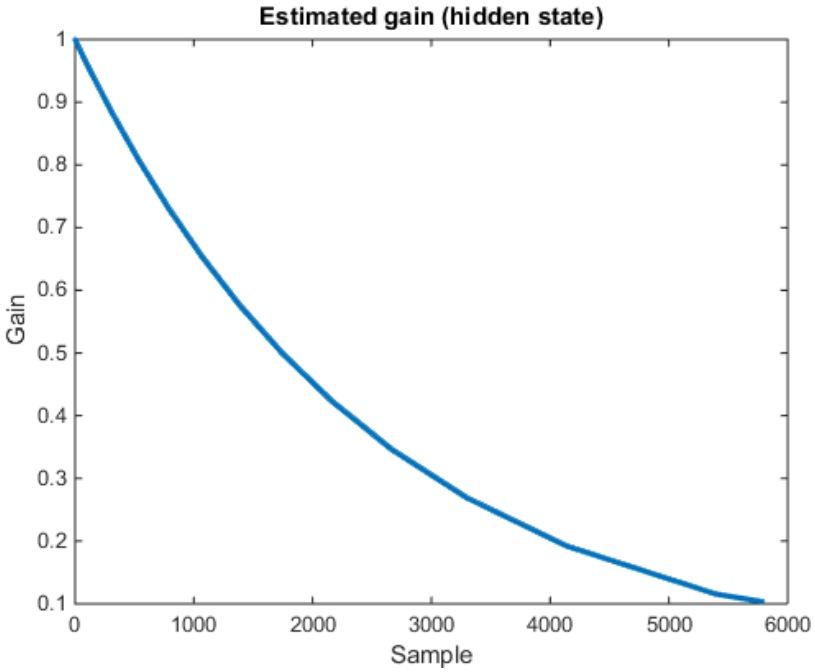


Figura 6.6 - Estimación Matlab de la ganancia del sensor.

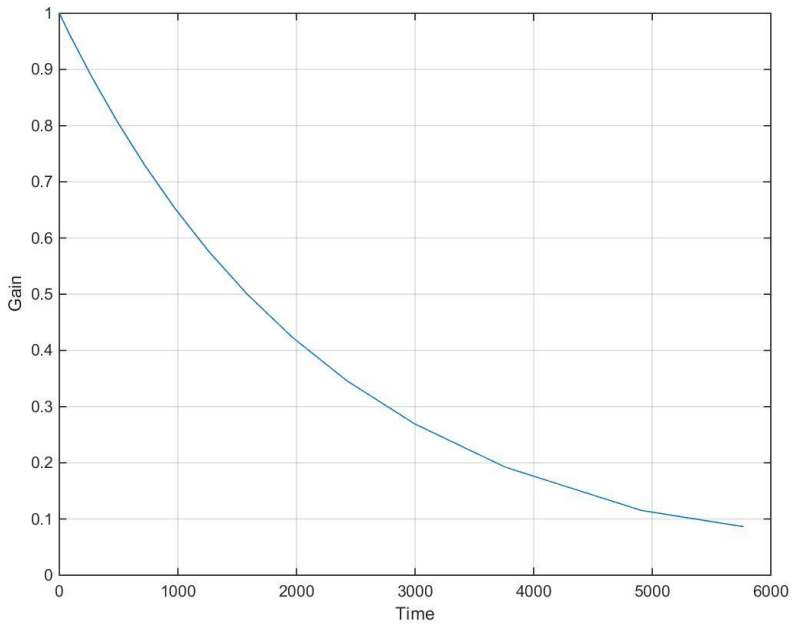


Figura 6.7 - Resultado de la ganancia del sensor con el testbench.

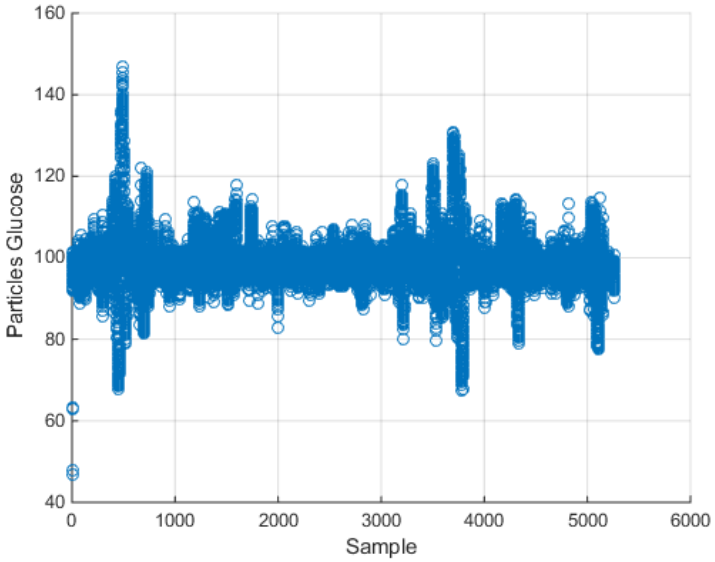


Figura 6.8 - Valores de glucosa para el grid de partículas.

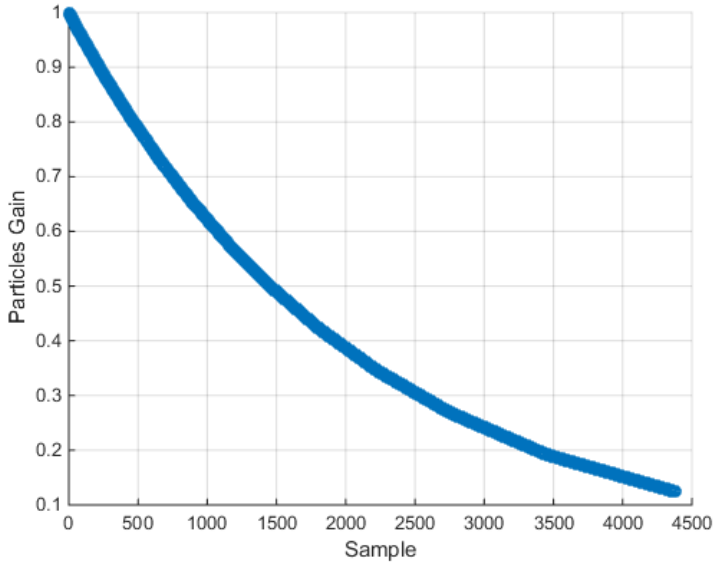


Figura 6.9 - Valores de la ganancia para el grid de partículas.

Con el fin de estudiar los picos en los resultados de la estimación de la glucosa, se realizó un estudio de los ruidos generados tanto en el diseño *hardware* como en el *script Matlab*. De esta forma, se analizó en primer lugar los ruidos aleatorios generados para la partícula 0. En la Figura 6.10 se observa el ruido generado en *hardware*, y en la Figura 6.11 se muestra el ruido generado desde el *script Matlab*. Tras analizar ambas imágenes se observa que los ruidos generados siguen un comportamiento muy similar, siendo la única diferencia que los valores generados en *hardware* se mueven en el rango [-8, 7], mientras que los generados en *Matlab* lo hacen en el rango [-7, 7].

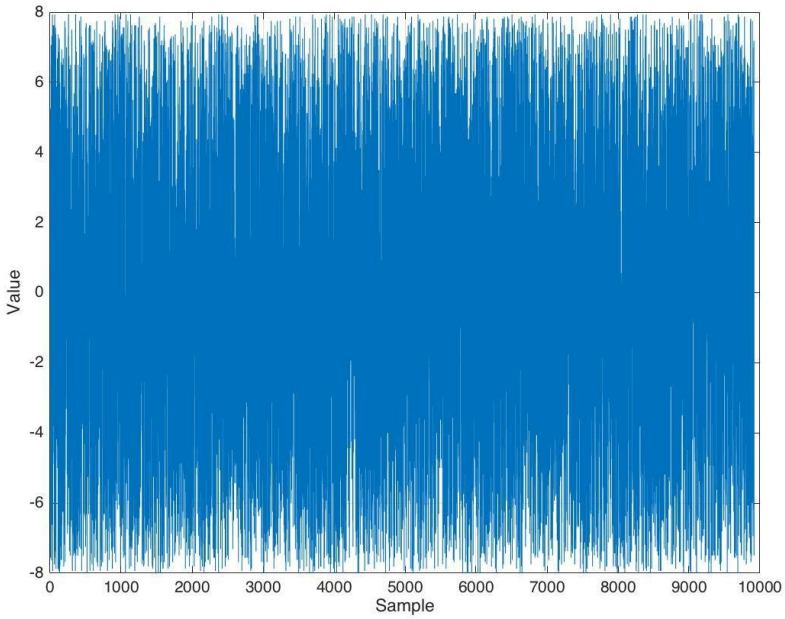


Figura 6.10 - Ruido generado para la partícula 0 en hardware.

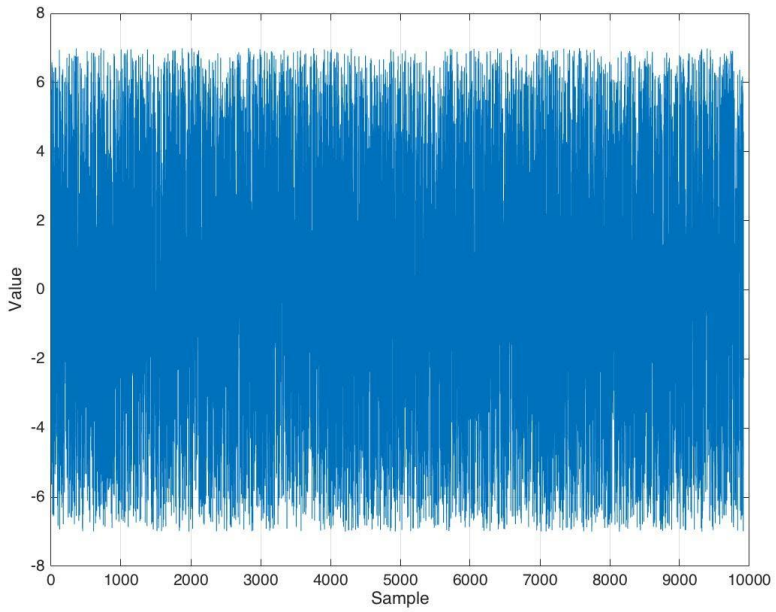


Figura 6.11Ruido generado para la partícula 0 en Matlab.

Debido a que los ruidos generados en ambas plataformas siguen una distribución similar, se trató de encontrar algún sesgo en el generador. De esta forma, se pasa a estudiar si el generador de números *pseudo*-aleatorios produce un tipo de número (positivos o negativos) con más frecuencia que otro. En caso de ser así, a largo plazo se produciría un desplazamiento en los valores del incremento de la glucosa bien hacia valores positivos o bien hacia valores negativos. Con el fin de ilustrar este comportamiento, se generaron dos imágenes representando el acumulado de los ruidos generados. En la Figura 6.12 se muestra el acumulado de los ruidos generados sobre *hardware*, se puede observar

que tienen una tendencia negativa. Esto se debe a que los rangos de los números generados no son simétricos [-8, 7]. Por otro lado, la Figura 6.13 muestra el acumulado de los ruidos generados en *Matlab*. En este caso, se observa que la suma de los ruidos oscila entre el valor 0, esto se debe a que los rangos, en este caso, sí son simétricos. Cabe destacar que las distribuciones de ambas representaciones resultan ser muy similares: ambas tienen picos positivos y negativos.

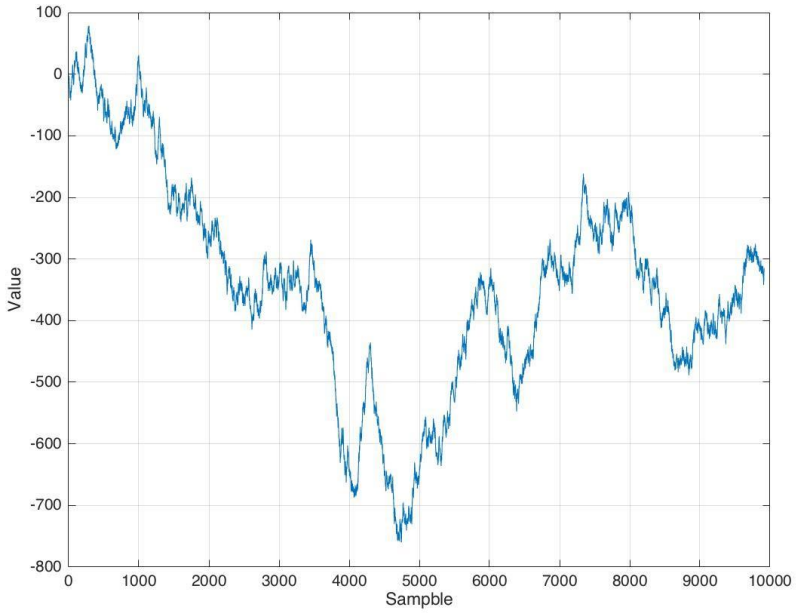


Figura 6.12 - Resultado de la acumulación de los ruidos en hardware.

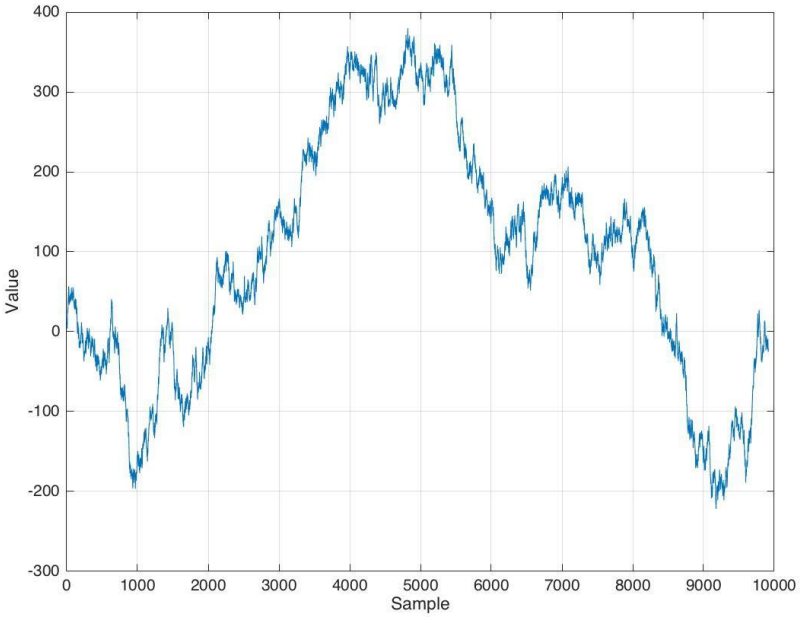


Figura 6.13 - Resultado de la acumulación de los ruidos en Matlab.

Por último, tras este estudio, se pasó a observar la evolución del incremento de la glucosa con el fin de conocer si el ruido generado provoca que este valor evolucione hacia números negativos. En la Figura 6.14 se observa cómo el incremento de la glucosa evoluciona tomando números negativos y positivos por igual. Por otro lado, este valor aumenta con el tiempo hasta saturar en valores en el rango [-8, 8].

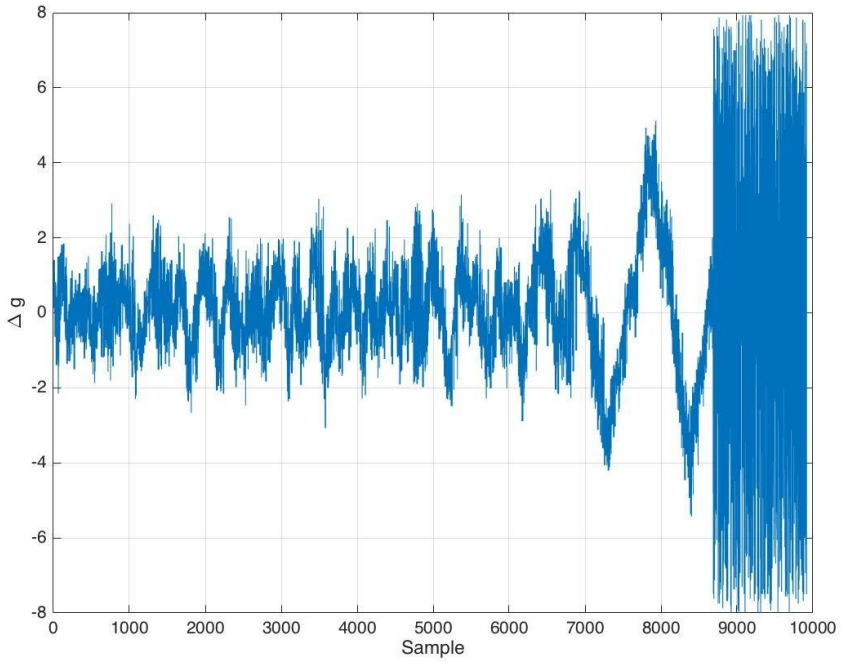


Figura 6.14 - Evolución del incremento de la glucosa en hardware.

Capítulo 7

7 Conclusiones y futuras líneas de trabajo

7.1 Conclusiones

Este trabajo forma parte de un proyecto de mayor magnitud que tiene como finalidad mejorar el control de la glucemia en pacientes con diabetes mellitus tipo 1. Actualmente este control se lleva a cabo mediante bombas de insulina y monitores continuos de glucosa. Estos últimos tienen incorporados sensores que miden los niveles de glucosa intersticial. Dichos sensores sufren una degeneración a lo largo de su vida útil, añadiendo así ruidos que es preciso eliminar. Por otro lado, no reflejan el valor auténtico de la glucosa en sangre, sino que obtienen la glucosa intersticial. La solución que se estudia en este trabajo es filtrar la señal del sensor con un filtro de partículas.

Este tipo de filtros estima el estado de un sistema dinámico a partir de un conjunto de medidas indirectas relacionadas con el mismo. En particular, utiliza un conjunto de estados probables del sistema (llamado *grid* de partículas) para reconstruir la función de densidad de probabilidad del estado y a partir de ella estimar la medida real del sistema.

Los filtros de partículas constan de cuatro etapas. En primer lugar, una etapa de inicialización, en la que se genera una primera población de partículas. Cada una de estas partículas tiene relacionado un peso que indica cómo de buena es dicha partícula. En segundo lugar, las otras tres etapas sobre las que se itera con el fin de evolucionar este primer *grid* de partículas:

- La etapa de predicción, en la que se aplican los modelos dinámicos del sistema y los modelos de medición. De esta manera se evolucionan los estados de cada partícula, y se estima la medición de glucosa en sangre.
- La etapa de corrección, donde se recalcula el peso para cada una de las nuevas partículas en función de la medida estimada y la medida real obtenida a través del MCG.
- Y, la etapa de *remuestreo*, donde se genera un nuevo *grid* de partículas del mismo tamaño que el inicial con aquellas partículas que tengan mejor peso.

En este proyecto se ha partido de un modelo de la evolución de la glucosa que tiene en cuenta la velocidad de variación de la misma y ciertos ruidos (tal y como se explicó en el capítulo 1). Lo mismo ocurre con los modelos de la evolución de la ganancia del sensor.

Cada una de estas etapas ha sido completamente diseñada en *VHDL*. El almacenamiento del *grid* de partículas sobre el que trabaja el filtro se guarda en un módulo llamado *particle_RAM*.

El filtro ha sido sintetizado y se ha generado un *bitstream* para su volcado sobre una *FPGA Virtex*-6 modelo *xc6vlx2t40*. El análisis estático de tiempos de esta síntesis estima que la frecuencia de trabajo máximo es 7.373MHz.

Por último, los resultados obtenidos tras la ejecución del *testbench* sobre el diseño VHDL del filtro han demostrado que es capaz de estimar una medida de glucosa con un valor constante de forma

satisfactoria. Sin embargo, en los resultados de la estimación de la glucosa se han observado ciertos saltos que, pese a que han sido estudiados, no se ha conseguido hallar el motivo de los mismos.

7.2 Futuras líneas de trabajo

En esta sección se van a abordar posibles futuras líneas de trabajo tal y como son:

- Implementar el método de la ruleta como método de remuestreo con el fin de estudiar si los saltos de la estimación de la glucosa son debidos al actual método de remuestreo.
- Realizar un *testbench post-síntesis* con datos de pacientes reales y comparar los resultados con los *scripts* desarrollados en *Matlab.*
- Desarrollar un *testbench* en *SystemVerilog* con el fin de verificar completamente el sistema.
- Realizar una comparación con el Trabajo Fin de Grado [10] realizado en el mismo departamento (*DACYA*) en el año 2015.

En primer lugar, se debería implementar en *hardware* el método de la ruleta explicado en la sección 2. Esto sería una idea interesante puesto que el actual método de remuestreo (método de torneo) escoge dos partículas al azar y selecciona la que mejor peso tenga. Sin embargo, se puede dar el caso de que dicho método escoja las dos peores partículas. De esta forma, es posible que se produjesen los saltos en la estimación de la glucosa. Es por ello, que mediante el uso de un método como la ruleta (en el que la partícula que más peso haya obtenido, tendrá una mayor probabilidad de ser seleccionada) se solucione dicho problema.

Por otro lado, sería necesario realizar un *testbench post-síntesis* para comprobar la correcta evolución del algoritmo en caso de contar con ruidos en el sistema. La intención consistiría en inyectar datos de glucosa de pacientes reales tanto al filtro *hardware* como al filtro *Matlab*. De esta forma, podrían compararse ambas salidas y verificar de esta forma el correcto funcionamiento del algoritmo *hardware*.

También sería interesante desarrollar una arquitectura de *testbench* utilizando el lenguaje *SystemVerilog*. Este lenguaje proporciona una gran variedad de herramientas para realizar *test* sobre este tipo de sistemas. De esta forma, se podría estudiar la cobertura tanto de código como de datos, buscando así un 100% en ambos tipos de coberturas. Es decir, demostrar que tanto todas las líneas del diseño han sido probadas, como que para todas las casuísticas de los datos el filtro se comporta como es esperado.

Y, por último, sería de gran interés comparar los resultados de este trabajo con el Trabajo de Fin de Grado [10]. Por un lado, podría compararse los resultados de la síntesis de ambos trabajos. Así, podría verse qué tipo de filtro ocupa más espacio, tiene un camino crítico más lento, consigue una frecuencia mayor, etc. Y, por otro lado, podrían compararse los resultados de la ejecución de ambos filtros y analizar cuál es más eficiente respecto a los datos calculados.

Capítulo 8

8 *Conclusions*

This work is a part of a larger project that aims to improve glycemic control in patients with diabetes mellitus type 1. Currently this control is carried out by insulin pumps and continuous glucose monitors. The later one have built sensors that measure interstitial glucose levels. These sensors suffer a degeneration throughout his life, adding noise that must be deleted. On the other hand, they do not reflect the true value of blood glucose. The solution that is studied in this work is to filter the sensor signal with a particle filter.

This kind of filters estimates the state of a dynamic system from a set of indirect measures related to it. In particular, it uses a set of probable states of the system (called grid particles) to rebuild the probability density function of the state and from it to estimate the real measurement of the system.

Particle filters consists of four stages. First, an initialization step, in which a first population of particles is generated. Each of these particles have a related weight indicating how good the particle is. Secondly, the other three stages on which is iterated in order to develop this first grid of particles:

- Prediction stage, in which the dynamic models of the system and measurements models are applied. Thus the states of each particle is evolving and the blood glucose is estimated.
- Correction phase, where the weight for each of the new particles is calculated according to the estimation and the actual measurement obtained through the MCG.
- And, resampling stage, where is generated a new particles grid with the same size as the initial with those particles that have better weight.

This project started from a model of the evolution of glucose which takes into account the rate of change of the same and has certain noises (as explained in Chapter 1). The same applies to the models of the evolution of the gain of the sensor.

Each of these stages have been completely designed in VHDL by project members. The particle storage grid on working the filter is stored in a module called particle_RAM.

The filter has been synthesized and generated a bitstream for FPGA Virtex dump on a xc6vlx2t40-6 model. The static timing analysis of this synthesis estimated maximum working frequency is 7.373MHz.

Finally, the results obtained after the implementation of VHDL testbench on filter design have shown that it is able to estimate a measure of glucose with a constant value satisfactorily. However, in the results of the estimation of glucose they were observed certain jumps that, although they have been studied, has not been successful in finding the reason for them.

Bibliografía

[1] R. A. d. l. l. Española, «Diccionario de la lengua española,» [En línea]. Available: http://dle.rae.es/?w=diccionario.

[2] Wikipedia, «Wikipedia - Diabetes Mellitus,» [En línea]. Available: https://es.wikipedia.org/wiki/Diabetes_mellitus#Diabetes_mellitus_tipo_1_.28DM-1.29.

[3] Sanjeev Arulampalam, Simon Maskell, Neil Gordon and Tim Clapp, «A tutorial on Particle Filters for On-line Non-linear/Non-Gaussian Bayesian tracking,» *IEEE TRANSACTIONS ON SIGNAL PROCESSING,* vol. 50, nº 2, 2002.

[4] Greg Welch and Gary Bishop, An introduction to the Kalman Filter, 2001.

[5] Thomas R. Bewley and Atul S. Sharma, «Efficient grid-based Bayesian estimation of nonlinear low-dimensional systems with sparse non-Gaussian PDFs,» *Automatica: A journal of IFAC the International Federation of Automatic Control, ISSN 0005-1098, Vol. 48, Nº. 7, 2012, págs.1286-1290,* p. 1, 2013.

[6] Wikipedia, «Wikipedia - Filtros Kalman Extendidos,» [En línea]. Available: https://en.wikipedia.org/wiki/Extended_Kalman_filter.

[7] A. D. a. S. J. Godsill, «Monte Carlo Filtering and Smoothing with application to time-varying spectral estimation,» *Acoustics, Speech, and Signal Processing, 2000. ICASSP '00. Proceedings. 2000 IEEE International Conference on,* 2000.

[8] Matthew Kuure-Kinsey, Cesar C. Palerm, B. Wayne Bequette, «A dual rate Kalman Filter for Continuous Glucose Monitoring,» *Engineering in Medicine and Biology Society, 2006. EMBS '06. 28th Annual International Conference of the IEEE,* pp. 63-66, 2006.

[9] Wikipedia, «Wikipedia - Equiprobabilidad,» [En línea]. Available: https://es.wikipedia.org/wiki/Equiprobabilidad.

[10] D. S. H. y. S. F. P. Fernando Capellán Pizarroso, «Diseño e implementación en FPGA de un filtro Kalman para aplicaciones biomédicas.».

[11] Mathwork, «Mathwork - Fixed point,» [En línea]. Available: http://es.mathworks.com/help/fixedpoint/.

[12] Mathwork, «Mathwork - LONGG Format,» [En línea]. Available: http://es.mathworks.com/help/matlab/ref/format.html.

[13] Dawn An, Joo-Ho Choi and Nam Ho Kim, «A tutorial for particle filter-based prognosis algorithm using Matlab,» *Reliability Engineering System Safety,* 2013.

[14] diabetes.org, «diabetes.org,» [En línea]. Available: http://www.diabetes.org/living-with-diabetes/treatment-and-care/blood-glucose-control/checking-your-blood-glucose.html?referrer=https://www.google.es/.

[15] D. P. B. a. V. I. Schellekens, «Field programmable Logic and applications,» *International Conference on,* pp. 1-6, 2006.

[16] M. P., «An analysis of random number generators for a hardware implementation of genetic programming using fpgas and handle-c,» *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '02, San Francisco, CA, USA. Morgan Kauf- mann Publishers Inc.,* pp. 837-844, 2002.

[17] P. Martin, An Analysis Of Random Number Proceedings of the Genetic and Evolutionary Computation Conference, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[18] D. a. P. B. a. V. I. Schellekens, Field Programmable Logic and Applications, 2006. FPL '06. International Conference on, 2006.

[19] Wikipedia, «Wikipedia - Regitro LFSR,» [En línea]. Available: https://es.wikipedia.org/wiki/LFSR.

[20] H. So, «Berkeley,» [En línea]. Available: http://www-inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html.